Principles
Of Operation

The EPSILON System

Version 1.0
15 June 1976

R.B. Talmadge

Is there anything whereof it may be said,
See this is new? It hath been already of
old time, which was before us.

Ecclesiastes 1:10

PREFACE

Principles of Operation, The EPSILON System, Version 1.0, was published on 15 June 1976 as an IBM Confidential document. It was declassified to non-confidential on 1 March 1978, with authorization to distribute copies to any interested party inside or out of IBM.

This document is a reprint of the original without revision of content. Minor text changes have been made, however, in order to take advantage of format improvements possible with new text and printing facilities.

CONTENTS

1.0 INTRODUCTION 1
- Nature of EPSILON Systems
- Relation to System/360/370
- References

2.0 BASIC CONCEPTS AND DEFINITIONS 4
- Processes
- Initiation and Termination
- Process Classes
- Storage
- Data Protection
- Instruction Sets
- Input/Output
- Addressing Conventions
- Instruction Execution
- Exceptions

3.0 STORAGE AND SPACES 13
- M-space Allocation
- B-space Allocation
- Pointers
- Space Retrieval
- Domain Identifier
- Changes of Custody
- Instruction Descriptions
  - ALLOCATE SPACE
  - FREE SPACE
  - SAVE SPACE
  - LOAD SPACE
  - STORE POINTER
  - LOAD POINTER
  - TEST POINTER
  - ASSIGN DOMAIN IDENTIFIER
  - LOAD DOMAIN IDENTIFIER
  - SET PROTECTION VECTOR
  - INSERT PROTECTION VECTOR

4.0 PROCESS DISPATCHING 28
- Computation Cycles
- C-process Dispatching Overview
- Dispatching Condition
- Selection Routines
- C-process Dispatching Example
- R-process Dispatching
- Computation Cycle Overrun

5.0 C-PROCESSES: GENERAL COMPUTATION 37
- Queues
- Process Model Definition
- Process Model Deletion

Initiation
State Vector
Process Entry
Linkage
Communication via Queues
Domain Identication
Shared Data
Deadlock Avoidance
Termination
Exception Handling
Forced Exceptions
Instruction Descriptions
DEFINE QUEUE
QUEUE INDEX
DEFINE C-PROCESS MODEL
STORE CMDB
DELETE QUEUE
INITIATE PROCESS
LOAD PROCESS INSTRUCTION COUNTER
IDLE
WAIT
CALL
RETURN
ENQUEUE
DEQUEUE
QUEUE SWITCH
QUEUE WAIT
CLOSE GATE
OPEN GATE
EXIT
TERMINATE PROCESS
DEFINE EXCEPTION MODULE
SET EXCEPTION MASK
SET BREAKPOINT MASK
FORCE PROCESS EXCEPTION

6.0 R-PROCESSES: EVENT RESPONSE 71
Process Sources
Process Model Connection
Initiation
Process Communication
Deadlock Detection
Termination
Exception Handling
Instruction Descriptions
STORE SOURCE LIST
CONNECT R-PROCESS MODEL
CONNECT R-PROCESS MODEL INDIRECT
STORE SOURCE STATUS
SIGNAL SOURCE

7.0 D-PROCESSES: INPUT/OUTPUT 85
I/O devices

Process Model Connection
I/O Requests
D-Process Environment
Dispatching
I/O Request Processing
Input/Output Operations
I/O Request Disposition
Use of Domain Identifier
Termination
Exception Handling
Attachment Interfaces
Instruction Descriptions
STORE DEVICE LIST
STORE DEVICE DESCRIPTION
CONNECT D-PROCESS MODEL
CONNECT D-PROCESS MODEL INDIRECT
STORE DEVICE STATUS
REQUEST INPUT/OUTPUT
TEST INPUT/OUTPUT
NEXT REQUEST
START DEVICE
WAIT DEVICE
HALT DEVICE
SET DEVICE STATUS
END I/O REQUEST
RESERVE DEVICE
END PROCESS

8.0 GENERAL INSTRUCTIONS 116
Fixed-Point Arithmetic
ADD
ADD HALFWORD
ADD LOGICAL
COMPARE
COMPARE HALFWORD
COMPARE LOGICAL
DIVIDE
LOAD
LOAD AND TEST
LOAD COMPLEMENT
LOAD HALFWORD
LOAD MULTIPLE
LOAD NEGATIVE
LOAD POSITIVE
MULTIPLY
MULTIPLY HALFWORD
SHIFT LEFT DOUBLE
SHIFT LEFT DOUBLE LOGICAL
SHIFT LEFT SINGLE
SHIFT LEFT SINGLE LOGICAL
SHIFT RIGHT DOUBLE
SHIFT RIGHT DOUBLE LOGICAL
SHIFT RIGHT SINGLE

SHIFT RIGHT SINGLE LOGICAL
STORE
STORE HALFWORD
STORE MULTIPLE
SUBTRACT
SUBTRACT HALFWORD
SUBTRACT LOGICAL
Logical Operations
AND
COMPARE LOGICAL
COMPARE LOGICAL CHARACTERS UNDER MASK
EXCLUSIVE OR
INSERT CHARACTER
INSERT CHARACTERS UNDER MASK
MOVE
OR
STORE CHARACTER
STORE CHARACTERS UNDER MASK
TEST AND SET
TEST UNDER MASK
TRANSLATE
TRANSLATE AND TEST
Branching
BRANCH AND LINK
BRANCH ON CONDITION
BRANCH ON COUNT
BRANCH ON INDEX HIGH
EXECUTE
LOAD ADDRESS
Long Operands
COMPARE LOGICAL LONG
MOVE LONG
TRANSLATE
TRANSLATE AND TEST
Decimal Feature
ADD DECIMAL
COMPARE DECIMAL
CONVERT TO BINARY
CONVERT TO DECIMAL
DIVIDE DECIMAL
EDIT
EDIT AND MARK
MOVE NUMERICS
MOVE WITH OFFSET
MOVE ZONES
MULTIPLY DECIMAL
PACK
SHIFT AND ROUND DECIMAL
SUBTRACT DECIMAL
UNPACK
ZERO AND ADD
Floating-Point Features
ADD NORMALIZED

ADD UNNORMALIZED
COMPARE
DIVIDE
HALVE
LOAD
LOAD AND TEST
LOAD COMPLEMENT
LOAD NEGATIVE
LOAD POSITIVE
LOAD ROUNDED
MULTIPLY
STORE
SUBTRACT NORMALIZED
SUBTRACT UNNORMALIZED

9.0 SERVICE PROCESSES 128
The System Clock
Time Events
System Exceptions
System Overrun
Invalid Process Model
Forced Exception
Statistics Collection
Domain End
Instruction Descriptions
REQUEST TIME EVENT
STORE CLOCK
FORCE SYSTEM EXCEPTION

10.0 SYSTEM INQUIRY FACILITIES 139
Configuration Data
Current State Data
Collection of Statistics
Assignment of Counters
Statistics Records
Domain Statistics
Software Statistics
Process Monitoring
Instruction Descriptions
STORE CONFIGURATION DATA
STORE PROCESS MODEL LIST
STORE DOMAIN LIST
STORE COMPUTATION CYCLE RECORD
COLLECT STATISTICS
SET COLLECTION MASK
DEFINE STATISTICAL COUNTER GROUP
ACQUIRE STATISTICAL COUNTER GROUP
SET MONITOR CONDITIONS
SET MONITOR MASKS

11.0 MALFUNCTION DETECTION AND RECOVERY 173
Hardware Malfunction
Invalid System Data

System Operation Error
Error Signal Mask
Error Signal Processes
Instruction Descriptions
SET ERROR MASK
DIAGNOSE

12.0 SYSTEM INITIALIZATION 180
Overview
Initialization Data Table
System Parameters
Dispatching Structure
Space Definition
Service Process Models
Regular Process Models
System Checkpoint
Application Initialization
System Termination
Instruction Descriptions
CHECKPOINT

1.0 INTRODUCTION

EPSILON is a computer system architecture which is an evolution of System/360 architecture into new control and addressing capability, leaving the basic computational capability unchanged. The primary objective is to provide a family of computer systems consistent with the installed S/360 base, that can be operated efficiently over a large set of configurations, hierarchically distributed or not, and over a substantially wider range of applications than current systems.

1.1 Nature of EPSILON Systems

In EPSILON systems many of the functions provided in other computer systems by supervisory control programs have been included in the basic instruction set, not only as a means of improving software performance but also as a way of promoting more uniformity in programming and operating systems. These supervisory functions, together with the facilities that support them, incorporate concepts and views of the computing environment which are sometimes exhibited explicitly, and are always inherent in the assumptions underlying the semantics of the instructions. The character and appearance of EPSILON systems is dominated by these concepts.

- Multiprocessing is assumed to be the rule rather than the exception, so that multiple concurrent processes are the expected norm. In order to provide a simple and stable programming environment in the face of the multitude of possibilities for processor interconnection and signalling, EPSILON systems explicitly recognize processes as the basic unit of organized instruction execution activity, and undertake to relieve software of dependency on physical processing mechanisms. Consequently, system microcode manages all activity connected with assignment of processors to processes, and the instruction execution behavior of an EPSILON system is independent of the number of processing mechanisms it contains.

- Two kinds of time constraints are subsumed as an integral part of the computing environment:

  - any event (signal) must be acted upon within some maximum time following its occurrence

  - any computation must be completed within a time interval determined by when the output of the computation is to be used.

  Because the time of occurrence of an event is unpredictable, event response constraints are in basic conflict with orderly computation, as they may require pre-emption of allocated resources. As an aid to the resolution of this conflict, processes in EPSILON systems are separated into three classes, each of which is provided facilities designed to match a particular kind of activity. One class is for general computation and two are for event response, one of those being for input/output. The system resolves basic resource conflicts between individual processes, and between process classes, using information supplied as part of on-going process activity.

- The general division of function between the system and individual processes is based on previous software experience, but the specific activity undertaken by the system is conditioned by a system view of the process class involved. Thus, process dispatching and storage allocation are functions of the system because experience has shown that they occur in nearly all general purpose supervisory software. However, dispatching of computational processes is time-driven, based on viewing those processes as data transformations, while dispatching of response class processes is event-driven.

  Similarly, storage is addressed only in segments (spaces) provided by the system in response to allocation instructions. Two levels of storage exist, one for data and instructions being processed, the other for permanent residence. Response class processes have only limited access to data in permanent storage, and none to instructions. Computational processes, however, are provided with automatic loading facilities that allow them to reference permanent storage directly in linkage instructions.

- The separation of process by class of activity leads to a natural separation of instructions by execution validity with respect to class, and to modal interpretation of instructions. An instruction may be executable only within a process of a given class, or may be common to two or more classes. Instructions common to more than one class may have the same interpretation for each class, or may have an interpretation which varies by class in order to be consistent with the presumed nature of the processes.

EPSILON systems therefore can be thought of as having three separate instruction sets, each of which presents programmers with a set of facilities designed to promote a particular class of activity. The instruction sets are independent in the sense that every process in the system consists of interpretation and execution of instructions selected from exactly one of the sets; they are connected in the sense that at least one instruction of each set can cause initiation of a process for some other set. The overall effect is one of interdependence, not because the instructions have the same formats (which is an accident of choice), but because every application is carried out by a mixture of processes from all three classes.

1.2 Relation to System/360/370

EPSILON data types are the same as those of System/370, have the same formats, and are subject to the same positioning rules with respect to half-word, word, and double-word boundaries. EPSILON instructions have formats identical to those of S/360 instructions, and where an instruction is common to EPSILON and S/360 the operation code is the same.

The EPSILON instruction list includes all non-privileged S/360 instructions, and all non-privileged instructions of S/370 with the exception of a few whose function is performed in some other way. Instruction interpretation is defined so that instructions executed within a single space of an EPSILON system, which refer only to data within that space, behave as they would when executed within a 360 or 370 system. Consequently, all S/360 and most S/370 programs which use only non-privileged instructions can execute without change on EPSILON

system models, and will obtain the same results as on S/360 or S/370 models.

The privileged instructions, the control registers, dynamic address translation, and other visible control mechanisms of S/360 and S/370 do not appear in EPSILON systems, as the EPSILON architecture extends S/360 in a direction for which such mechanisms are inappropriate. Furthermore, the input/output system, while similar to S/360 and S/370 at the device command interface, operates within a disciplinary framework not required on those systems.

A microcode feature is available which will provide for interpretation of S/360 privileged instructions. A similar feature is separately available for S/370. These features each provide a basic mechanism by which any S/360 or S/370 program can execute within a single space of an EPSILON system, but it is expected that software will provide the additional functions required for correct interpretation of the program within its programming system environment.

1.3 References

This document is intended as a self-contained principles of operation. However, to avoid duplication of much that is well-known, the reader is assumed to have a basic knowledge of S/360, and some material is included by reference to S/360 and S/370 principles of operation:

- POP360 indicates a reference to SRL document GA22-6281-8, IBM System/360 Principles of Operation, ninth edition, November 1970

- POP370 indicates a reference to SRL document GA22-7000-3, IBM System/370 Principles of Operation, fourth edition, January 1973.

References are enclosed in square brackets when they are ancillary to the text, but are not enclosed when they supply the text. External references include a colon and page number, other references do not. Thus, [POP360:41] refers to page 41 of the S/360 principles of operation, and [Section 4.3] refers to Section 4.3 of this document.

2.0 BASIC CONCEPTS AND DEFINITIONS

The following paragraphs describe some of the basic system concepts and introduce definitions which will be used throughout the remainder of the document. The defined terms, including ordinary words used in a special technical sense, appear in **bold** type at their first appearance. If a term can have values, the first appearance of the name of a value is either underlined or exhibited on a separate display line.

2.1 Processes

All instruction execution in an EPSILON system must occur as part of an organized activity called a **process.** A process consists of a succession of states, each of which is described by the data contained in a **state vector.** Transition from a given state to its successor state is accomplished by action of some processing mechanism, which

- selects an instruction from the sequence of instructions currently associated with the process
- interprets the instruction in the context of the state vector data, and carries out the function of the interpreted instruction
- alters the state vector data (if necessary) to reflect the result of execution of the instruction.

The number of processes active in the system at any time is arbitrary, limited only by availability of physical resources. The number of processing mechanisms in the system is not a factor which limits the number of processes; in fact, the functional (i.e. instruction execution) behavior of the system is not affected by the number of processing mechanisms, though the system throughput or response to events may well be affected.

2.2 Initiation and Termination

Process initiation is the activity of bringing a process into being and setting up its initial state. Initiation can occur only as the result of activating a **process source,** and is carried out as a closed function of the system. Some process sources are specifiable as part of the system and are factory or field installed, others are generated as a result of instruction execution. The following kinds of process sources occur or may occur in any EPSILON system.

- **Input/Output Devices.** A source is installed at the factory or in the field for every I/O device port attached to a system. An I/O device source is activated whenever an associated device signals a device event (e.g. attention). I/O device sources can also be activated by the REQUEST INPUT/OUTPUT instruction.
- **External Signal Interface.** An external signal interface can be factory or field installed in any EPSILON system. A source is installed for each signal line, and is activated when the appropriate signal appears on the line.
- **Queues.** Queues can be specified to act as process sources. The source is activated whenever an item is placed into the queue and the queue was empty at the time.

Activation of a process source always results in a request for process initiation. The request may not result

in the actual initiation of a process, however, if a process already exists which meets the request specifications.

Process termination is the activity of deleting a process from the system; it is the inverse of process initiation, and is also carried out as a closed function of the system. Termination can occur only as the result of execution of a termination request instruction, either by the process to be terminated or by a process detecting a requirement for termination.

2.3 Process Classes

In order for process initiation to operate as a closed function, it is necessary to provide the system with data which describes the structure and content of the initial state vector. Such a collection of data is called a **process model**, and a process derived from a given model is called an **instance** of that model. All processes are instances of some process model.

There are three classes of process model, each giving rise to a corresponding class of process. The classes are designated as the **C-process** class, the **R-process** class, and the **D-process** class. The capital letters used to distinguish these process classes will also be used to distinguish items and characteristics unique to the classes. Thus, 'D-process model' will be used to refer specifically to a process model giving rise to a process of the D-process class, while 'process model' will be used if the reference is to apply to any process model.

Each of the three classes of process is designed to provide facilities which are matched to a particular class of activity:

- C-processes for general computation
- R-processes for response to external or internal event signals
- D-processes for I/O device control and data transfer.

However, any process can carry out any kind of activity as long as it has access to the instructions and data required for that activity.

The separation of process models and processes into classes is achieved by adoption of rules and procedures which regulate the general character of their behavior. Aspects regulated include:

- **Initiation.** Process sources are restricted to association with specific classes. I/O device sources, for example, can cause initiation of D-processes or R-processes, but not C-processes, while queues can only cause initiation of C-processes.
- **Instruction set.** The classes differ in the states allowed their processes. Consequently,
  - the structure and content of the associated state vectors is not the same from class to class
  - the instructions which can be executed vary by class.

  A given processing mechanism is therefore not necessarily assignable to more than one class.
- **Process management.** The process dispatching rules, the amount and type of dispatching control, and the communication and data sharing mechanisms made available to processes, distinguish the process management functions

of one class from those of another.

The rules and procedures governing behavior of processes appear in Chapters 5, 6, and 7, which discuss the specific characteristics of each process class.

2.4 Storage

Main storage in an EPSILON system consists of two collections of data bytes, called **M-storage** and **B-storage**. M-storage has predictable access time, and may be volatile; B-storage has unpredictable access time, and is non-volatile. Data and instructions may occupy B-storage for permanent residence, but must reside in M-storage before they can be processed.

Initially, both types of storage consist of a pool of 8-bit bytes which are unrelated and not addressable. The total number of bytes of storage and the number of bytes in each pool is installed at the factory, but additional bytes may be field installed. Addressability is obtained by executing an allocation instruction, which defines a unit of storage, either an **M-space** or a **B-space**, consisting of a specified number of bytes withdrawn from the M-storage or B-storage pool.

- Both M-spaces and B-spaces are referenced by use of a 32-bit identifier, called a **pointer**, unique to the space. The pointer for a space is assigned by the system when the space is allocated. Pointer values are not predictable, except for the null pointer which is always a zero field.

- Bytes within a B-space are not individually addressable, though they are considered to be sequenced contiguously, starting with sequence number 0.

- Bytes within an M-space are addressable, with address locations corresponding to sequence numbers. A 24-bit address is applied relative to location 0 to reference any desired byte. Addressing therefore accommodates M-spaces up to 16,777,216 bytes in extent.

A space remains in existence until a FREE SPACE instruction referring to it is successfully executed, at which time it is deleted from the system and the M-storage or B-storage pool enlarged by the number of bytes in the M-space or B-space. The total number of spaces in existence at any one time is constrained only by the requirement that the sum of all M-space extents not exceed the total number of M-storage bytes installed in the system, and the sum of all B-space extents not exceed the total number of B-storage bytes installed.

Information can be moved between M-storage and B-storage by executing

- a LOAD SPACE instruction, which copies the contents of a B-space into an M-space in byte sequence order

- a SAVE SPACE instruction, which copies an M-space into a B-space.

However, there are no instructions which will alter the division of bytes between M-storage and B-storage.

2.5 Data Protection

In order to provide a basic framework for protection of data, every space is assigned to the **custody** of some process or group of processes, and allocated read and write **access**. Custody identifies the group of processes that can delete the space or

change its access. There are three types of custody.

- **Private.** The space can be deleted or have its access type altered only by one specific process.
- **Family.** All processes which are instances of a given process model are said to be members of that process model **family.** Family custody allows any member of the family to delete the space or alter its access type.
- **Bound.** The space is bound to some process model, or to the system. It cannot have its access type altered, and can be deleted only when the process model is deleted, or the system is initialized.

Custody is transferred by the execution of certain instructions. For example, when an ENQUEUE instruction is completed, the space is removed from its current custody and becomes part of the queue. It is subsequently assigned to the custody of the process which dequeues it.

The read access of a space identifies the group of processes that can read data from the space, and the write access those that can store data into it. In decreasing order of restriction, the types of access are

- Private: access restricted to a single process
- Family: access restricted to members of a particular family
- Domain: access restricted to processes acting on behalf of some specified data domain [Section 3.5]
- Public: access allowed to any process.

The **protection vector** of a space is the triple of the values of custody, read access, and write access. M-spaces can have any legitimate protection vector value. B-spaces, however, cannot be assigned private custody or private read access. These restrictions are automatically applied by the instructions which act on B-spaces.

2.6 Instruction Sets

Processing mechanisms are either internal to an EPSILON system or are devices attached externally by means of the I/O attachment interfaces. I/O devices (or device adapters) execute instructions which are specialized to their particular control and data transfer characteristics. They execute such instructions as part of a D-process, being assigned as the associated processing mechanism on execution of the START DEVICE instruction.

In contrast to these external device instruction sets, all internal EPSILON instructions conform to specific formats and prescribed interpretation conventions. They are classified by execution validity with respect to process class, and by mode of interpretation.

- Some instructions are executable within a process of any process class, some are common to two of the process classes, some are valid only within one of the classes.
- The instructions valid within R-processes are called the **basic instruction set,** those valid within C-processes are called the **computational instruction set,** and those valid within D-processes are called the **peripheral instruction set.** A **decimal feature** and a **floating-point feature** may be independ-

ently added to the standard computational instruction set.

- Instructions common to more than one process class may have a fixed interpretation, or may be **modal** instructions whose interpretation varies with class. Modal instructions allow subroutines to be written which automatically conform to the differing interpretation conventions of the process class within which they are executed.

Internal processing mechanisms are of two types with respect to the instruction set classification.

- A **central processing unit** (CPU) can execute the basic instruction set or the computational instruction set, or both.

- A **peripheral processing unit** (PPU) can execute the peripheral instruction set.

There must be at least one CPU and one PPU in every EPSILON system. If there is only one CPU it must have both the basic instruction set and the standard computational instruction set installed; if there is more than one CPU, then the division between basic and computational instruction capability is arbitrary, as long as each capability is installed. Some models, however, may offer only limited combinations of CPU capability. The decimal and floating-point features can be installed only in a CPU in which the computational instruction set has been installed.

2.7 Input/Output

An input/output operation transfers data between an M-space and an I/O device. The transfer mechanism is provided by the peripheral processing units of the system, each of which actually consists of two parts:

- an **I/O attachment interface**, which supplies logical and electrical connection between M-storage and I/O devices or device adapters

- a processing mechanism for interpretation and execution of the instructions by which D-processes control transmission of data between M-spaces and I/O devices.

The attachment interface of a PPU is either a DC interlocked interface, called a **channel**, or a serial, clocked, bit-frame transmission interface, called a **loop**. The processing mechanism of a PPU interprets instructions and data the same way in either case, so that processes are not affected by the form of attachment chosen for any I/O device.

2.8 Addressing Conventions

The EPSILON instructions have the same structure and the five basic formats of S/360 instructions: RR, RX, RS, SI, and SS [POP360:12,13]. There are, however, significant differences of interpretation of the fields referenced in the instructions:

- register references designate special data private to a process

- storage addresses are generated by rules which cause reference to a location in some allocated space.

The state vector of every process contains sixteen 8-byte fields, called **general registers**, which are referenced by the R, X, and B fields of any instruction executed within the process. The state vector of every C-process also contains four additional 8-byte fields, called

floating-point registers, which are referenced by the R field of floating-point instructions executed within the process.

The eight bytes of a general register are divided into two independent 4-byte fields. The first field, called an **arithmetic register**, always contains a binary number used either for arithmetic calculation or to compute locations relative to the origin of some allocated space. The second field, called a **pointer register**, can contain only a pointer to an M-space or B-space, or the null pointer, and is used only for address generation.

In the RR, RX, and RS instruction formats, the R field may designate the arithmetic register, the pointer register, or the complete general register. In the RX instruction format, the X field designates the arithmetic register, treated as a 24-bit index. In the RX, SI, and SS instruction formats, the B field designates the complete general register, whose arithmetic and pointer fields combine to form the base for an operand address. Address generation is carried out as follows.

- The contents of the pointer register designated by the B field identifies the space being addressed. An access exception [Section 2.10] will occur if the register does not contain a pointer to an allocated M-space or B-space.

- The low order 24 bits of the contents of the arithmetic register designated by the B field are treated as an unsigned binary integer specifying the **base address** relative to the identified space.

- The low order 24 bits of the contents of the arithmetic register designated by the X field (in RX format instructions) are treated as an unsigned binary integer **index** relative to the base address.

- The 12-bit number contained in the D field of the instruction is treated as an unsigned binary integer **displacement** relative to the base address.

- The full address is calculated by adding the base address, index, and displacement as 24-bit binary numbers, ignoring overflow, to form a 24-bit location value. This value designates the byte in the identified space whose sequence number is equal to the location value.

In forming addresses, a special interpretation is given to zeros in the X and B fields, and to the null pointer. A zero in the X field indicates the absence of indexing, and a zero will be used for the index value irrespective of the contents of arithmetic register zero. A zero in the B field is treated in the same way if the address being generated is actually a location relative to an implied space, as in the LOAD ADDRESS, EXECUTE, and branching instructions [Section 8.3]. No special treatment is applied to a B field of zero when the space must be explicitly identified, as is the case with most instructions.

The null pointer is interpreted as referring to a special null space which has public read and write access, and is zero bytes in extent. An address always lies outside the null space, so that an addressing exception will occur if the address is generated in connection with any instruction which refers to a location in the space.

2.9 Instruction Execution

When a processing mechanism is assigned to a process, it fetches instructions from the M-space location designated by the process instruction counter contained in the state vector of the process. The instruction address is then increased by the number of bytes in the instruction in order to address the next instruction, except that in the case of branching, linkage, and loop control instructions, the next instruction address is calculated as part of the instruction execution itself.

In concept, the processing mechanism fetches and executes instructions one at a time, with the results of the execution of one instruction available preceding the execution of the next instruction. In practice, instructions may be fetched out of order or executed in a sequence physically different from the conceptual one, in order to take advantage of overlap of instruction fetch with operand access. However, the results generated for any instruction are those that would have been generated if the conceptual sequence had been followed, as a processing mechanism will not be switched from one process to another without first having brought physical and conceptual execution into agreement.

These actions also provide assurance that data private to a process will appear to behave exactly as expected by the conceptual execution sequence. In the normal course of events, however, there is no such implicit assurance that data accessible to more than one process will appear to behave with such integrity, as all processes in an EPSILON system can be advancing concurrently. This occurs not only because there generally are multiple CPU and PPU, but also because any particular processing mechanism may be switched from one process to another between any two instuctions. Consequently, the instructions CLOSE GATE and OPEN GATE provide controlled access to shared data, for use when the logic of a process requires that an instruction sequence have exclusive use of the data for some period of time [Section 5.10].

2.10 Exceptions

When an instruction is completed, a 2-bit **condition code** field in the state vector of the process may be set to a value which indicates an attribute of the result. For example, the condition code is set to the value 1 for a fixed-point addition with a negative result, and to a 2 for a positive result. The condition code is inspected by the BRANCH ON CONDITION instruction, which then uses the value of the code to select the next instruction address.

If a condition is encountered during instruction execution which precludes the expected result, it may be indicated in the condition code, or a **process exception** may be raised. In general, the condition code is used to indicate unusual conditions not under control of the process within which the instruction is being executed (e.g. the 'storage not available' condition for ALLOCATE), while an exception signals a condition for which the process itself is responsible.

There are six exceptions which can occur for improper specification or use of an instruction, one which can be forced, and two which arise from fixed-point arithmetic. These exceptions, and their significance, are as follows.

- Operation: Either the operation code of the instruction is not assigned, the instruction is not

installed, or the instruction cannot be executed within the process because of process class execution restrictions

- Execute: An EXECUTE instruction is the subject of an EXECUTE instruction

- Access: Either a storage reference is invalid because the pointer does not identify an allocated space, or the space has access for the instruction not allowed to the process

- Addressing: A generated address lies outside the extent of the referenced space, or the space is currently not available for addressing

- Specification: An operand of the instruction does not meet some requirement restricting its location, reference identifier, or length

- Data: An operand of the instruction does not meet some requirement on its structure or value

- Forced: Occurs as the result of executing a FORCE PROCESS EXCEPTION instruction, or when a process trace record is stored

- Fixed-point overflow: A high-order carry has occurred or high-order significant bits have been lost as a result of executing a fixed-point add, subtract, shift, or sign-control instruction

- Fixed-point divide: Either fixed-point division by zero has been attempted, the quotient exceeds the register size, or the result of a CONVERT TO BINARY instruction exceeds 31 bits.

In addition, the following exceptions can occur when the decimal and floating-point features are installed.

- Decimal overflow: The destination field of a decimal instruction is too small to contain the result

- Decimal divide: The quotient of a decimal instruction exceeds the size of the specified data field

- Exponent overflow: The result of a floating-point add, subtract, multiply, or divide instruction has a non-zero fraction and a characteristic greater than 127

- Exponent underflow: The result of a floating-point add, subtract, multiply, divide, or halve instruction has a non-zero fraction and a negative characteristic

- Significance: The result of a floating-point addition or subtraction has a zero fraction

- Floating-point divide: A floating-point division by zero has been attempted.

An 8-bit field in the state vector, called the **exception mask** field, determines the treatment of the eight arithmetic exceptions. Each bit of the mask corresponds to one of the exceptions; if the bit is 0, the exception will be ignored if raised, if the bit is 1 the exception will be signalled to the process. The other exceptions cannot be masked, so are always signalled to the process. The methods of signalling exceptions vary by process class, as do the facilities for acting in response to them. Details of exception procedures appear in Chapters 5, 6, and 7.

Unusual conditions can also arise

which are not directly relatable to execution of a particular instruction. Dispatching, for example, detects a possible CPU overload as a result of trying to meet the CPU requirements of all processes [Section 4.7]. Conditions of this kind for which remedial action is possible raise a **system exception**, and cause an exception process to be initiated. Exception process models are built-in to the system, but are personalized to individual systems by data supplied at system initialization. Details of system exceptions and system exception procedures appear in Chapter 9.

3.0 STORAGE AND SPACES

In any EPSILON system, storage is withdrawn from the storage pools to contain data and data structures used for system management. Part of the storage is withdrawn at system initialization [Chapter 12], while the remainder is withdrawn as the result of conditions which occur during system operation (e.g. storage is required for state vector data for a newly initiated process). Once withdrawn, such storage is not returned to the storage pools, but is retained and managed separately by the system microcode.

The amount of storage used by the system thus grows with demand, but the demand falls off sharply as the data structures adjust in size to the operational load. After a relatively short initial period, the demand for additional system storage will occur only during periods of extraordinary use. These periods will themselves occur with decreasing frequency, so that eventually the division between system storage and storage available for space allocation remains constant. The value of this steady-state constant cannot be predicted with certainty, but it is possible to determine an approximate value from system initialization data.

Once the steady state has been reached, the execution time of all instructions which involve direct or indirect storage allocation falls within the bounds prescribed in the functional specifications of the individual EPSILON systems.

3.1 M-space Allocation

The ALLOCATE SPACE instruction (ALLOC) provides for direct allocation of two kinds of M-spaces:

- **ordinary M-spaces** are assumed to be intended to hold data but not instructions to be executed; such spaces cannot be the object of a linkage instruction
- **module M-spaces** are assumed to be intended to hold instructions to be executed. A module space can be addressed to store and fetch data, but cannot be enqueued or transferred outside the custody of the original process family, except to bound custody; it can be made available to other processes as the object of a linkage instruction.

The attribute of being a module or ordinary space is permanently retained by an M-space, and is inherited by any B-space or M-space allocated as a descendant of the space.

The amount of space to be allocated is specified as an exact number of bytes. The system may choose to allocate storage in multiple-byte units in order to simplify internal mechanisms. The amount of space allocated may therefore not be exactly the same as that requested, and the difference may vary from one EPSILON system to another. However, in all cases the amount actually allocated will not be less than that requested.

3.2 B-space Allocation

B-space allocation is always indirect in the sense that it occurs only in connection with saving data in an M-space by means of the SAVE SPACE instruction (SAVE). SAVE is a modal instruction which can be executed within a C-process or an R-process. When executed within a C-process, it is entirely synchronous with respect to process state advancement. When

the instruction is completed:

- a B-space equal in size to the referenced M-space has been allocated
- the number of bytes in the space and the space pointer have been loaded into the arithmetic and pointer register fields of the specified general register
- the contents of the referenced M-space have been stored into the allocated B-space.

When executed within an R-process, SAVE is not entirely synchronous. It is completed as soon as the B-space has been allocated and the general register loaded; storage of the M-space data into the allocated B-space may not be complete, nor even in-process. Until data storage is complete, the newly allocated B-space is not available, and an addressing exception will occur if it is referenced prior to completion. The LOAD POINTER instruction indicates space availability, and so can be used to avoid premature references.

The B-space allocated by a SAVE instruction inherits the attribute of being an ordinary or module space from the saved M-space. It also inherits the protection vector, though some values may be altered.

- The M-space may be in private or family custody of the process executing the SAVE instruction; the B-space is always put into custody of the family of the process
- if the read or write access of the M-space is private, the corresponding B-space access is set as family; other access values are inherited unaltered.

Although a B-space is assigned write access, the access is significant only for M-space descendants of the space, as there are no instructions which copy data into an existing B-space. A **descendant** of a B-space is an M-space allocated during execution of a LOAD or linkage instruction referencing the B-space, or a B-space allocated during execution of a SAVE instruction referencing an M-space descendant of the B-space, or any space allocated during the execution of a LOAD, linkage, or SAVE instruction referencing a descendant of the B-space.

The LOAD SPACE instruction (LOAD) is a modal instruction matched to the SAVE instruction. When executed within a C-process, it behaves like a SAVE instruction executed within a C-process with the roles of the B-space and M-space reversed. Thus, when the instruction is completed:

- an M-space equal in size to the referenced B-space has been allocated
- the number of bytes in the space and the space pointer have been loaded into the arithmetic and pointer register fields of the specified general register
- the contents of the referenced B-space have been copied into the allocated M-space.

Similarly, when executed within an R-process, it behaves like a SAVE instruction executed within an R-process with the roles of the B-space and M-space reversed. In either case, the descendant M-space inherits without change both the protection vector of the B-space and its atribute of being an ordinary or module space.

A LOAD instruction can be applied to any B-space, whether an ordinary or

module space, and a new descendant M-space results from each application. New B-spaces can then be allocated by application of SAVE instructions to the M-spaces, so that descendant trees of any degree of complexity can be formed by use of the SAVE and LOAD instructions.

A linkage instruction, however, can only be applied to a B-space with the module attribute, and a new descendant results only if one does not already exist; thus, only one M-space copy of a module B-space need exist at any one time. The linkage instructions are discussed in Chapter 5.

3.3 Pointers

Pointers are generated only when new spaces are allocated by the instructions ALLOC, SAVE, and LOAD. A new pointer is loaded into the pointer register designated in the allocating instruction, where it is available for use by the process within which the instruction was executed. Two instructions exist to make pointers generally available.

- The STORE POINTER instruction (SP) stores the contents of the designated pointer register into a word located in an M-space, where the pointer can be retrieved by any process having access to the space.

- The LOAD POINTER instruction (LP) loads the contents of a word located in an M-space, presumed to be a pointer, into a designated pointer register.

Because a space cannot be referenced except through a pointer in a pointer register, the LP instruction is designed to indicate space availability, and to serve as the focal point for validation of access. The status of the space relative to the process is returned in the condition code set by the instruction, so that processes can avoid references which would cause exceptions.

- Condition code zero indicates the process has both read and write access to the space

- Condition code 1 indicates the process has access restricted to read or write

- Condition code 2 indicates the space exists but is temporarily unavailable to the process

- Condition code 3 indicates the space is not available to the process, either because all access is denied or the space does not exist.

In order to allow optimization of instruction execution performance, the status defined by an allocation instruction or returned by an LP instruction is also retained by the system as the status associated with the use of the pointer register. The retained status is not changed if the relation between the process and the space changes (e.g. the space becomes available for addressing) until another LP referencing the register is executed. A process can therefore gain access to a space if it was previously denied only by explicitly loading a pointer to the space into some pointer register.

If a process has loaded a pointer and received an indication of space availability, it is still possible for access or addressing exceptions to occur if the space is not of the right type for the instruction. When there is doubt, the type may be determined by the TEST POINTER instruction (TP), which returns data indicating what type of space is identified by the contents of a specified pointer register.

3.4 Space Retrieval

A space is automatically deleted from the system whenever its custodian is deleted:

- a space in private custody is deleted during termination of the custodian process
- a space in family custody is deleted during deletion or modification of the process model for the family
- a space in bound custody is deleted with the process model to which it is bound.

No explicit deletion request is required of any process for such deletion to occur. A space can also be explicitly deleted by execution of a FREE SPACE instruction (FREE) within any custodian process of the space.

A legitimate deletion request, whether indirect or explicit, will always be accepted for any space. However, if the space is not available for addressing because a previous instruction has not been completed, or if the space contains a closed access control gate [Section 5.10], deletion will be delayed until the space becomes eligible to return to normal status, without the return actually being made.

Deletion will also be delayed until processes which have gained access to the space no longer need to reference it. For this purpose, reference requirements are measured in terms of pointer register usage. It is presumed that a process will expect to continue to reference a space as long as a pointer granting access to the space resides in a pointer register of the process. Consequently, a **reference count** is recorded and maintained for each space.

- The count is set to the value 1 when the space is allocated, representing the usage of the pointer register loaded by the allocation instruction. Allocation also sets a **custody flag** for the space. The space cannot be deleted as long as the custody flag is turned on.
- The count is incremented by 1 whenever a pointer granting access to the space is loaded into a pointer register of some process; it is decremented by 1 whenever a pointer to the space is deleted from a pointer register in some process. An LP instruction completed with condition codes zero or 1 will therefore cause the count of the space referred to by the pointer just loaded to be incremented, and the count of the space referred to by the pointer displaced from the register, if any, to be decremented.
- A FREE instruction substitutes a null pointer for any pointer to the referenced space in all pointer registers of the process executing the FREE, and the reference count is decremented by 1 for each substitution. The custody flag is also reset, indicating loss of custody.
- During process termination counts are decremented by carrying out the equivalent of loading a null pointer into all pointer registers of the process. Custody flags are reset by deletion requests generated for spaces in private custody of the process.

A space is not actually deleted from the system until its custody flag is turned off and its reference count becomes zero. If a deletion request is accepted and the count does not go to zero at acceptance, references to

the space not involving data transfer are treated as if the space did not exist:

- an LP instruction will return condition code 3
- all instructions requiring a process to have custody of the space, such as LOAD and SAVE, will cause exceptions or return a condition code indicating invalid usage, as appropriate to the instruction.

Deletion therefore always appears synchronous to the request. When a space is finally deleted, the storage associated with the space is returned to the M-storage or B-storage pool, as appropriate.

Reference count protection is also applied to other instructions which result in transfer of custody, such as ENQUEUE [Section 5.8] and REQUEST INPUT/OUTPUT [Section 7.3].

3.5 Domain Identifier

A **domain** is an unordered, non-empty collection of ordinary spaces. It begins existence with a single space, acquires new members through process activity, loses members as they are deleted from the system, and goes out of existence if all of its member spaces are deleted. The period of existence of a domain is not fixed, nor is there a limit to the number of members, either in total or at any given time.

The criteria for membership in a domain are not explicitly defined, so there is no information about the content of a space belonging to a given domain which is made use of by the system. Domains are simply recognized as entities for which membership is propogated by rules relating spaces and processes.

- Each domain has a **domain name** and **domain identifier.** The name is an arbitrary 32-bit referent supplied as data; the identifier is a 32-bit referent assigned by the system. The name is used only when a domain is formed, and is returned when the domain goes out of existence. All other references are by means of the identifier.
- A domain is formed by the ASSIGN DOMAIN IDENTIFIER instruction (ASSIGN), which generates a new domain identifier and assigns it to a specified space. The space must be an ordinary space in custody of the process executing the ASSIGN, and the name must be distinct from all other domain names, or the assignment attempt will be rejected.
- As a matter of convenience, a space which does not belong to a named domain is considered to belong to an unspecified domain called the **common domain.** The value of the identifier (and also the name) of the common domain is zero; the value of other domain identifiers is not predictable.
- The activity of any process is always considered to take place on behalf of the domain whose identifier has been acquired by the process. A process may acquire a permanent identifier or may be assigned a succession of identifiers. The specific rules for acquisition of identifier, which vary by process class, are given in Chapters 5, 6, and 7. In all cases, when a process with a given domain identifier allocates an ordinary space, the domain is propogated by assigning that identifier to the allocated space. A module space is always assigned to the common domain.

- The domain identifier assigned to a space is retained until the space is deleted, unless the space is the object of an ASSIGN instruction and becomes the initial space of a new domain. Any B-space or M-space allocated as a descendent of a space inherits the identifier current at the time of allocation. The identifiers of descendents are not altered by an ASSIGN applied to a space.

- The **membership count** of a domain is set to the value 1 by a successful ASSIGN, incremented by 1 for every space added to the domain, and decremented by 1 for every space removed from the domain. If the count goes to zero, the domain is deleted from the system and a 'domain end' system exception is raised. The exception process is supplied the resource usage statistics logged against the domain identifier [Section 9.8].

As it is possible to allocate any I/O device so that I/O requests will be accepted only from processes with a specified identifier [Section 7.9], these facilities allow domain identification to be used as a basis for the definition, control, and accounting of work within an EPSILON system. As additional support for this function, the domain identifier of a space may be obtained by use of the LOAD DOMAIN IDENTIFIER instruction (LDID), which places the identifier in the arithmetic register field of a designated general register.

3.6 Changes of Custody

The ALLOC instruction also provides options for the initial setting of the protection vector. It is possible to request that the allocated M-space be assigned either to private custody of the process within which the allocation instruction was executed, or to custody of the family of that process. The same request can also be made for the read and write access, so that the initial protection protection vector of an M-space can have any one of the eight values from

(private,private,private)
.
to
.
(family,family,family).

The custodian process can then change the custody or access of the space by executing either a SET PROTECTION VECTOR instruction (SPV), or an ENQUEUE instruction.

The SPV instruction provides the means for direct alteration of the protection vector of an M-space or a B-space. The new protection vector of the space is the result of applying the maximum function between the old protection vector and the proposed vector. The maximum is taken component by component, using numerical values for custody and access types of

0 = private
1 = family
2 = domain
3 = public.

SPV operation is therefore always towards less restrictive protection for the space. Thus, the initial application of SPV to a space allows the custody to be enlarged to that of the family to which the custodian process belongs, and the read or write access to be set to any of the values family, domain, or public, but subsequent applications may have no effect at all.

The ENQUEUE instruction provides the means to transfer custody of an M-space from one family to another,

although the transfer is indirect in the sense that the family associated with the queue is not known to the enqueuing process. Execution of an ENQUEUE will cause transfer of custody to whatever process dequeues the space, at which time the protection vector will be set as if the space had been newly allocated by direct request of the dequeuing process.

Once an M-space has been placed into family custody, any member of the family has custodian status, and so can execute an ENQUEUE or SPV without causing an access exception. Since the SPV does not provide any way to change custody from family to private, the space can return to private custody only if transferred by means of ENQUEUE.

Access, custody, and domain membership are related in the following way.

- A space with private access can be referenced for data access of that type (read or write) only by the original custodian process. If the space is transferred to family custody without enlarging the access to family, the other custodians cannot reference the space for that type of access. If the original custodian is deleted, no process can then have that type of access unless custody is transferred to another family.

- A space with family access can be referenced for data access of that type by any member of the family.

- A space with domain access can be referenced for data acces of that type only by a process whose domain identifier matches the identifier of the space. A custodian process has no special status in this case.

- A space with public access can be referenced for data access of that type by any process.

- A module space can be referenced by a process for instruction execution only if the process has read access to the space.

For purposes of access a space in bound custody is treated as if it were in custody of the family of the process model to which it is bound, though processes of the family are not, in fact, custodians of the space. Neither SPV nor ENQUEUE can be used to place a space into bound custody. A space becomes bound to a process model when the model is defined; once bound, it cannot be returned to private or family custody.

3.7 Instruction Descriptions

The following are detailed descriptions of the function and behavior of the storage and space instructions. These descriptions are in the same format as the instruction descriptions in POP360, except that the instruction picture and operation code have been omitted, and a category has been added specifying the process classes within which the instruction can be executed. To save repetition, the following exception descriptions common to all instructions have been omitted from the text, and are not specifically listed under the heading 'Exceptions':

- if an attempt is made to execute an instruction by a process belonging to a process class which is not allowed execution rights, the instruction is suppressed with an operation exception

- if an attempt is made to reference a space through a pointer register with an invalid access status for the type of reference,

the instruction is suppressed with an access exception

- if a pointer register expected to designate an M-space (B-space) actually contains a pointer to a B-space (M-space), the instruction is suppressed with a specification exception.

Suppression of an instruction causes the exception to be taken before the state vector is altered or data in storage has been modified.

ALLOCATE SPACE

ALLOC M1,R2 <RR>

The M-storage pool is examined for a contiguous block of storage of extent not less than the number of bytes specified in arithmetic register R2. If the request exceeds the total amount of available M-storage installed, the instruction is terminated with condition code 2. If the request could be met but no block of sufficient size is currently available, the instruction is terminated with condition code 1.

If a block is available an M-space is allocated to fulfill the request. The custody flag of the space is turned on, and the reference count is set to the value 1. If the space is an ordinary space, it is assigned the domain identifier of the process within which the instruction is being executed. If the space is a module space, it is assigned to the common domain. The membership count of the assigned domain is incremented by 1.

The four bits of the mask field M1 specify attributes of the allocated space. If the high-order bit is 1 the space is designated as a module space, otherwise it is an ordinary space. The three remaining bits indicate initial assignments for protection vector values, corresponding in high to low order respectively to custody, read access, and write access. If a bit is zero, the corresponding position of the protection vector is set to private, referring to the process within which the instruction is executed; if the bit is 1, the corresponding position is set to family, referring to the family of that process.

The actual extent of the allocated space, which is not less than that requested, then replaces the contents of arithmetic register R2, a pointer to the space is loaded into pointer register R2, and the instruction is completed with condition code zero.

Process Class: C,R,D

Condition Code:

0	Space Allocated
1	M-storage not currently available
2	Request cannot be fulfilled
3	-

Exceptions: None

FREE SPACE

FREE R2 <RR>

The space designated by pointer register R2 is deleted from the system and the storage associated with the space returned to the storage pool corresponding to the type of space deleted.

If the requesting process is not a custodian of the space the instruction is suppressed with an access exception. It is suppressed with a data exception if the space is in I/O request state [section 7.3].

If the instruction is not suppressed, the custody flag of the space is turned off. A null pointer is loaded into pointer register R2 and into any other pointer register of the process which contains a pointer to the space, and the instruction is completed by decrementing the reference count by 1 for each null pointer loaded.

Deletion of the space will occur when both the reference count and the gate count [Section 5.11] are zero, which may be prior to completion of the instruction or at some later time. When the space is deleted, the membership count of the domain to which it belongs is decremented by 1. If the count becomes zero, the domain is deleted from the system and a domain end system exception is raised. In any event, an attempt by any process to load a pointer to the space after the custody flag has been turned off will be rejected. References to data in the space by addresses generated through pointer registers loaded before the flag was turned off will be valid until the space is actually deleted.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions:
- Access
- Data
- Domain end (system)

SAVE SPACE

SAVE R1,R2 <RR>

The B-storage pool is examined for a contiguous block of storage of extent not less than the number of bytes in the M-space designated by pointer register R1. If the request exceeds the total amount of available B-storage installed, the instruction is terminated with condition code 3. If the request could be met but no block of sufficient size is currently available, the instruction is terminated with condition code 2. If the requesting process is not a custodian of the designated M-space, the instruction is suppressed with and access exception.

If a block is available, a B-space is allocated to fulfill the request. The newly allocated space inherits the protection vector, custodians, domain

identifier, and ordinary or module attribute from the referenced M-space. The protection vector of the B-space is then examined, and any values of private are changed to values of family. The extent of the space, which is at least that of the referenced M-space, is placed into arithmetic register R2 and a pointer to the space is placed into pointer register R2. The reference count of the space is set to the value 1, its custody flag is turned on, and the membership count of the domain is incremented by 1.

If the instruction is executed within a C-process, the contents of the M-space are then copied into the newly allocated B-space, and the instruction is completed with condition code zero.

If the instruction is executed within an R-process, ■ request is made to copy the contents of the M-space into the newly allocated B-space, and the instruction is completed with condition code 1. The B-space is not available for addressing until data transfer from the M-space is complete, and an addressing exception will occur if it is prematurely referenced. The availability status can be tested by loading the space pointer into a pointer register. The contents of the B-space are not predictable if the M-space contents are altered after completion of the instruction but prior to completion of the data transfer.

Process Class: C,R
Modal

Condition Code:
0 Space saved
1 Space allocated
2 B-storage not available
3 Request cannot be fulfilled

Exceptions:
Access

LOAD SPACE

LOAD R1,R2 <RR>

The M-storage pool is examined for a contiguous block of storage of extent not less than the number of bytes in the B-space designated by pointer register R1. If the request exceeds the total amount of available M-storage installed, the instruction is terminated with condition code 3. If the request could be met but no block of sufficient size is currently available, the instruction is terminated with condition code 2. If the requesting process is not a custodian of the designated B-space, the instruction is suppressed with an access exception.

If ■ block is available, an M-space is allocated to fulfill the request. The newly allocated space inherits the protection vector, custodians, domain identifier, and ordinary or module attribute from the referenced B-space. The extent of the space, which is at least that of the referenced B-space, is placed into arithmetic register R2 and a pointer to the space is placed into pointer register R2. The reference count of the space is set to the value 1, its custody flag is turned on, and the membership count of the domain is in-

cremented by 1.

If the instruction is executed within a C-process, the contents of the B-space are then copied into the newly allocated M-space, and the instruction is completed with condition code zero.

If the instruction is executed within an R-process, a request is made to copy the contents of the B-space into the newly allocated M-space, and the instruction is completed with condition code 1. The M-space is not available for addressing until data transfer from the B-space is complete, and an addressing exception will occur if it is referenced prematurely. The availability status can be tested by loading the space pointer into a pointer register.

Process Class: C,R
Modal

Condition Code:
0 Space saved
1 Space allocated
2 M-storage not available
3 Request cannot be fulfilled

Exceptions:
Access

STORE POINTER

```
SPR  R1,R2  <RR>
SP   R1,D2(X2,B2)  <RX>
```

The contents of pointer register R1 are stored into the word at the second operand location.

In the RR form of the instruction, the second operand is arithmetic register R2. Pointer register R2 is not disturbed.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions: None

LOAD POINTER

```
LPR  R1,R2  <RR>
LP   R1,D2(X2,B2)  <RX>
```

The instruction is suppressed with a specification exception if the format of the second operand is not consistent with a pointer. It is terminated with condition code 3 if the space is not available to the process because it does not exist, is being deleted, or the process does not have access to it, and with condition code 2 if the space is temporarily unavailable to the process.

If the space is available to the process, pointer register R1 is examined and if it contains a pointer the reference count of the identified space is decremented by 1. If decrementing causes the count to become zero, if the custody flag is off, and if the gate count is zero, deletion of the space will occur as described for the FREE SPACE instruction.

The second operand is then loaded into R1 and the reference count of the identified space is incremented by 1. In the RR form of the instruction, the second operand is arithmetic register R2.

The instruction is completed with condition code zero if the process is allowed both read and write access to the space, and with condition code 1 if access is restricted to read or write only.

Process Class: C,R,D

Condition Code:

0	Full access granted
1	Restricted access granted
2	Space temporarily unavailable
3	Space not available

Exceptions:

Specification
Domain end (system)

TEST POINTER

TP R1,D2(X2,B2) <RX>

The instruction is terminated with condition code 3 if pointer register R1 contains a pointer to a space temporarily unavailable to the process within which the instruction is being executed, and with condition code 2 if the register contains a null pointer.

If the instruction is not terminated, a byte of descriptive data is stored at the location designated by the second operand. The instruction is then completed with condition code zero if the space is an M-space, and with condition code 1 if the space is a B-space.

The byte of stored data describes the relation of the space to the process. The high-order bit is zero if the space is an ordinary space, and 1 if it is a module space. Bit 1 is zero if the process is not a custodian of the space, and 1 if it is; bit 2 is zero if the process does not have read access to the space, and 1 if it does; bit 3 is zero if the process does not have write access, and 1 if it does. The remaining bits of the byte are set to zero.

Process Class: C,R,D

Condition Code:
0 M-space data stored
1 B-space data stored
2 Null space
3 Space temporarily unavailable

Exceptions: None

ASSIGN DOMAIN IDENTIFIER

ASSIGN R1,R2 <RR>

The instruction is suppressed with an access exception if the requesting process is not a custodian of the space designated by pointer register R2, and with a specification exception if the space is not an ordinary space.

If the instruction is not suppressed, the contents of arithmetic register R1 are compared with the list of domain names. If there is a match, the instruction is terminated with condition code 1. If there is no match, an identifier is generated for a new domain with the specified name. The identifier replaces the contents of arithmetic register R1.

The membership count of the new domain is set to the value 1. The new domain identifier replaces the domain identifier previously assigned to the space, and the membership count of the old domain is decremented by 1. If decrementing reduces the count to zero, the domain is deleted from the system and a domain end system exception is raised. The instruction is then completed with condition code zero.

Process Class: C,R

Condition Code:
0 Identifier assigned
1 Domain already exists
2 -
3 -

Exceptions:
Access
Specification
Domain end (system)

LOAD DOMAIN IDENTIFIER

LDID R1,R2 <RR>

The domain identifier of the space designated by pointer register R2 replaces the contents of arithmetic register R1.

Process Class: C,R

Condition Code: Unchanged

Exceptions: None

SET PROTECTION VECTOR

SPV M1,R2 <RR>

If the requesting process is not a custodian of the space, the instruction is suppressed with an access exception. Otherwise, the protection vector of the space designated by pointer register R2 is modified according to the contents of arithmetic register R2, under control of mask field M1.

The high-order bit of field M1 is ignored. The three remaining bits indicate assignments for protection vector values, corresponding in high to low order respectively to custody, read access, and write access. If a bit is zero, the corresponding position of the protection vector is to remain unaltered; if a bit is 1, the position is altered as determined by arithmetic register R2.

The bytes of the register, numbered 0,1,2,3 from left to right, correspond to the bits of field M1. Byte 0 is therefore ignored, and bytes 1, 2, and 3 contain values for custody, read access, and write access. The low-order bit of byte 1 and the two low-order bits of bytes 2 and 3 specify the request as

0 = private
1 = family
2 = domain
3 = public

For any position of the vector which is to be altered, the new value assigned corresponds to the numerical maximum of the existing value and the requested value.

Process Class: C,R

Condition Code: Unchanged

Exceptions:
Access

INSERT PROTECTION VECTOR

IPV R2 <RR>

The protection vector of the space designated by pointer register R2 is inserted into arithmetic register R2.

The high-order byte of the register is set to the value zero if the space is an M-space, and to the value 1 if the space is a B-space. The three remain-

ing bytes correspond in high to low order respectively to custody, read access, and write access. The protection vector components are inserted into the two low-order bits of the corresponding bytes using the values

0 = private
1 = family
2 = domain (access) or bound (custody)
3 = public

The four high-order bits of each of the bytes are set to zero. The instruction is completed with condtion code zero if the space is an M-space, and with condition code 1 if it is a B-space.

Process Class: C,R

Condition Code:

0	M-space vector inserted
1	B-space vector inserted
2	-
3	-

Exceptions: None

4.0 PROCESS DISPATCHING

During its lifetime, a process is either advancing from state to state because a processing mechanism is acting upon the instruction sequence associated with the process, or else it is suspended in some state waiting for assignment of a processing mechanism. The activity involved in the assignment of a processing mechanism to processes is called **process dispatching.**

Apart from the option of defining scheduling algorithms at system initialization, dispatching is carried out by the system as a closed function, using data supplied in process models and data reflecting the current status of the system. Dispatching discipline varies with process class; for C-processes, dispatching is essentially time-driven, while for R-processes and D-processes it is event-driven. This difference is typical of the different kind of activity visualized for the process classes.

This chapter describes CPU assignment, and the relation between C-process dispatching and R-process dispatching. D-process dispatching is discussed in Chapter 7.

4.1 Computation Cycles

A C-process is viewed by the system as an activity which transforms data from a given input form to some specified output form. The process may exist only for a single transformation, but in general its life will extend over a series of transformations. A simple and natural way to describe the CPU requirements of such processes is in terms of time constraints for completion of a transformation. For example, if the function of a particular process is to receive messages and distribute them to other processes on the basis of decoding a field in the message, and if ten messages a second can be expected, then the process must be assigned a CPU often enough to complete an average of one transformation every 100 milliseconds.

Such a description is intrinsic to the process and independent of other processes, but provides no basis for relating process requirements to one another, as the transformation carried out by a process is chosen arbitrarily. However, any period of time shorter than an intrinsic time constraint will fulfill the same requirement, and any such shorter time can be chosen to be a multiple of some fixed unit of time. These observations form the basis for the introduction of time into C-process dispatching.

- The fixed unit of time is called a **basic cycle.** The length of the basic cycle is specified at system initialization, and can be changed only by re-initialization. Its value must be larger than the time taken to execute the dispatching function by that model of EPSILON system, but is otherwise not restricted.

- The time constraint of any process is specified in terms of a length of time called a **computation cycle.** The number of computation cycles and the length of each cycle is also specified at system initialization.

The computation time, T, alloted to each computation cycle is a multiple of the basic cycle time, t. That is,

$$T = Nt.$$

The integer N, which can range between 1 and 16,777,215, is called the **period** of the computation cycle. Each cycle is also assigned a unique 8-bit identifier, so that a system may have as many as 256 separate computation cycles. The association of a process with a computation cycle then occurs as follows:

- every C-process model designates a computation cycle by specification of its identifier
- all members of a process family are assigned to the computation cycle identifed by their process model.

There are no instructions which will change the assignment of a process, once initiated, from one computation cycle to another. Change can only affect processes not yet initiated, as the result of changing the process model association through use of the DEFINE C-PROCESS MODEL instruction.

4.2 C-process Dispatching Overview

The overall structure of C-process dispatching is represented by the schematic of figure 4.1.

The objective of this structure is to furnish enough function to provide a useful service, but not so much function as to preclude any algorithm for apportionment of CPU time considered equitable by managers of a system.

The basic concept behind the structure is to treat computation cycle periods as time constraints for selecting processes to be allocated a CPU. Every process assigned to a given computation cycle is to be selected, or considered for selection, once every period of the cycle. Figure 4.1 can therefore be visualized as the flow of control through a continuing activity of the system governed by those time constraints, and with the following general behavior.

- Dispatching activity is triggered whenever a condition arises which forces a process switch (e.g. an I/O wait), and at the beginning of every basic cycle. The basic cycle is therefore the maximum time that can elapse between attempts to assign a CPU to a C-process.
- The function of entry service is to initiate a process to service overrun of the computation cycle time constraints, should it occur, and to serve as a control point to idle if there is no process to dispatch.
- If there has been any request for initiation of a C-process since the previous cycle of dispatching activity, initiation service is invoked. Its function is to generate initial state vector data and to assign the resulting process to the appropriate computation cycle. As the process model may not have been in recent use, and the initial instruction sequence is not required to be pre-loaded into an M-space, initiation of a particular process can require a complex sequence of actions extending over many cycles of dispatching activity [Section 5.4].
- Selection of a process, which is called **scheduling**, consists of first selecting a computation cycle, then a process within the cycle. For this purpose, the computation cycles are sequenced in order of period, from smallest to largest. Initially, the cycle with the smallest period is chosen and processes are selected from it. Processes continue to be selected from that cycle until an indication is given by the scheduling mechanism that the

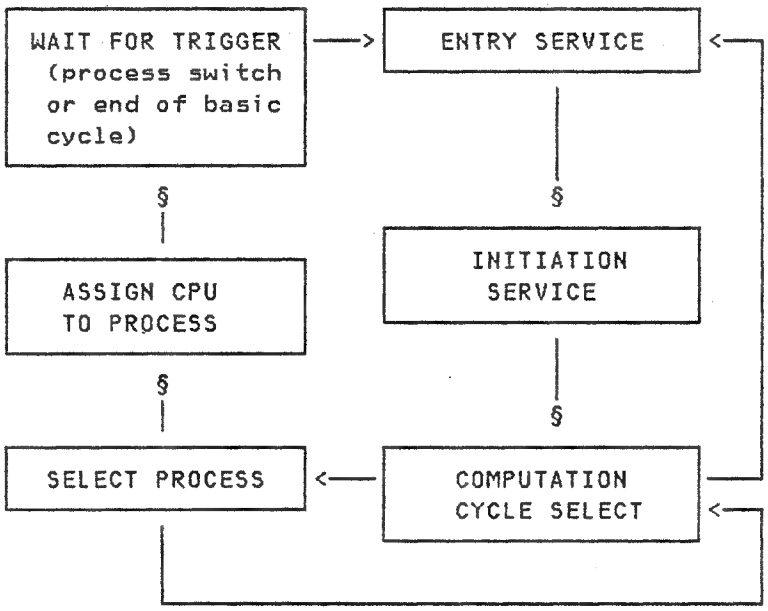

Figure 4.1
Schematic of C-process
Dispatching Activity

list of processes assigned to the cycle is exhausted. If time then remains before expiration of the period, the next cycle in sequence is chosen. If the period expires before the process list is exhausted, a **computation cycle overrun** condition may exist; in that event, selection remains with the same cycle rather than proceeding to the next.

Selection activity continues in this way until all computation cycles are exhausted, which causes dispatching to idle, or until expiration of a basic cycle, which forces re-consideration of the initial computation cycle. In passing from one cycle to the next, if the new cycle has been previously completed, it will not be re-started again until expiration of its current period. Furthermore, a given cycle will not be reached until all previous cycles have completed their current periods. The overall effect of this procedure is to multiplex CPU allocation for all computation cycles preceding any given cycle within each period of that cycle.

- Since an optimal selection procedure is quite dependent on application mix, the algorithm used to select a process within a computation cycle is not fixed. Each computation cycle has an associated microprogram routine, called a **selection routine**, which implements the scheduling algorithm for that cycle. Selection routines are specified at system initialization, either as one of three standard routines, or as a feature routine.

  It is the selection routine of a computation cycle which determines

- what process has been selected, if any

- whether or not the process list is exhausted.

As a routine is not restricted in the algorithm by which it determines these conditions, the significance of any computation cycle period is actually defined by the selection routine for the cycle.

- Once a process has been selected, a CPU is assigned to it. Assignment consists of loading the state vector of the process into the local storage and registers of the CPU, causing execution of the currently associated instruction sequence to resume from the point of previous suspension. Dispatching activity then idles until another entry trigger occurs.

Execution of the instruction sequence of a selected process will continue until a process switch is required, or until a basic cycle expires. The average duration of process activity therefore depends primarily on the instruction sequence and CPU speed, so that expression of intrinsic time constraints in terms of computation cycle periods is normally a direct conversion.

4.3 Dispatching Condition

Although a selection routine may implement any scheduling algorithm in principle, in practice its behavior is constrained by the dispatching **condition** of the processes which are candidates for selection. The condition of a C-process is:

- **running**, if a CPU is assigned to it

- **waiting**, if some event must occur before instruction execution can proceed

- **ready**, if neither running nor waiting.

The running and ready conditions arise out of dispatching action, but waiting conditions result from instruction execution. There are two kinds of waiting condition. An implicit wait occurs when instruction interpretation is delayed until some condition is met, at which time the instruction will be completed. When that occurs, the process is said to be **suspended**. An explicit wait is one which results from a condition generated by completion of an instruction. Waiting conditions are further subdivided into five classes.

- **I/O Wait.** The process is waiting for completion of a LOAD, SAVE, CALL or REQUEST INPUT/OUTPUT instruction.

- **Gate Wait.** The process is waiting for completion of a CLOSE GATE instruction, giving it access to data under control of an access control gate.

- **Exception Wait.** The process is suspended until completion of an exception process initiated by a FORCE SYSTEM EXCEPTION instruction.

- **Queue Wait.** The process is waiting for data to be entered into a specified queue. The wait was initiated by execution of a QUEUE WAIT instruction, and will end with arrival of the data.

- **General Wait.** The process is waiting for an unspecified occurrence. The wait was initiated by execution of a WAIT instruction; it will end whenever

an initiation request is recorded for the process.

Assignment of a CPU to a process only makes sense, therefore, if the process is in ready condition. So if the basic operation of a scheduling algorithm results in picking a process that is running or waiting, the selection routine should take action to select another process, and should continue selecting until a ready process is found or the computation cycle list is exhausted. All useful scheduling algorithms obey this constaint.

4.4 Selection Routines

Standard selection routines are supplied with all EPSILON systems which implement simple scheduling algorithms of fairly wide applicability.

Finite Sequential

Processes in the computation cycle are selected in a sequence determined by a sequencing number supplied in the process model. The first process is selected at the beginning of a period, the second at next entry, the third at next entry, and so on. If a process is not ready at selection, it is bypassed in favor of the next in sequence. Once selected or bypassed, a process will not be considered again until the next period, so the routine returns a list exhausted indication when the last process in sequence has been examined.

This algorithm amounts to a round-robin each period, with slack time in the period filled in by processes from computation cycles of larger period.

Infinite Sequential

Process sequence is determined as in the finite algorithm; however, selection starts at the first in sequence at every entry, the process chosen being the first one encountered in ready condition. A list exhausted condition is never returned unless there is no ready process in the computation cycle.

This algorithm is the same as one often called 'priority scheduling'. It blocks access to computation cycles of larger period unless the cycle is dormant, in effect extending the selection period indefinitely beyond the nominal cycle time. The expected usage is for the last computation cycle, where 'background' processes, if they exist, would normally be placed.

Multiplexed

Processes are assigned a weighting factor, supplied in the process model, which specifies the relative proportion of the computation cycle to be assigned to the process. The algorithm selects a process as many times in a cycle as the proportionality factor indicates, spreading the selections as evenly as possible throughout the period. An indication to use the next cycle for one selection is returned whenever a selected process is not ready or when slack time occurs in the period.

This algorithm is a weighted round-robin, with slack time spread throughout the period and filled in by processes from computation cycles of larger period.

All three selection routines treat processes in a gate wait condition as a special case, in order to apply a promotion technique to empty the gate

queue as rapidly as possible. This technique is discussed in Section 5.10.

4.5 C-process Dispatching Example

In a particular system there are three computation cycles, one of period 2, one of period 3, and one of period 5. The finite sequential algorithm is used for the first two cycles, and the infinite sequential for the last. Process 1 is assigned to the first cycle, processes 2 and 3 to the second, process 4 to the last.

Figure 4.2 illustrates one of the possible flows of control if the system has a single CPU. The bottom line represents time divided into basic cycles, with numbers marking the end of each cycle. The three upper lines show allocation of the CPU to processes in the respective computation cycles. Pointed right brackets designate the beginning of a computation cycle period, and colons within a period indicate computation cycle selection.

In this example

- Process 1 was not assigned a CPU during the fourth period of its computation cycle because it returned a WAIT during basic cycle 5 which was not resolved until after basic cycle 8 had started.

- The CPU becomes idle during basic cycle 6 when process 4 enters I/O wait, and stays idle until the next computation cycle period starts.

- Process 2 is time-sliced at the end of basic cycle 1 and is not assigned the CPU again until the next period; process 3 is treated the same way at the end of basic cycle 7. Process 4, however, having come out of I/O wait during basic cycle 7, is assigned the CPU during basic cycle 8 becaue of the behavior of the infinite sequential algorithm.

Figure 4.3 illustrates a control flow for the same case if the system has two CPU. In this figure, two sets of three lines are used to show allocation of the CPU to processes in the respective computation cycles. The first set of lines shows allocation of the first CPU, while the second set corresponds to the second CPU. The time-slicing behavior of the finite algorithm is more apparent here.

This simple example is designed to show the multiplexing effect of computation cycle dispatching on C-process execution. It does not show the possibility of computation cycle overrun, nor the effect of R-process dispatching.

4.6 R-process Dispatching

An R-process is viewed by the system as an activity which reacts to the occurrence of some recognized event. The process is expected to exist only as long as necessary for the initial response, and to cause a C-process to be invoked to carry out any additional activity which may be required. A natural way to describe the CPU requirements of such processes is in terms of response priorities, where higher priority is equated to greater urgency. Consequently, every source of R-processes is assigned a priority with respect to the other sources. The assignment is set up during system initialization, and cannot be changed by instruction execution.

It is also a consequence of this view that the initiation of an R-process coincides with the initial assignment of a CPU, and that a CPU switch will occur only if required to service a higher priority event. Instruction behavior within

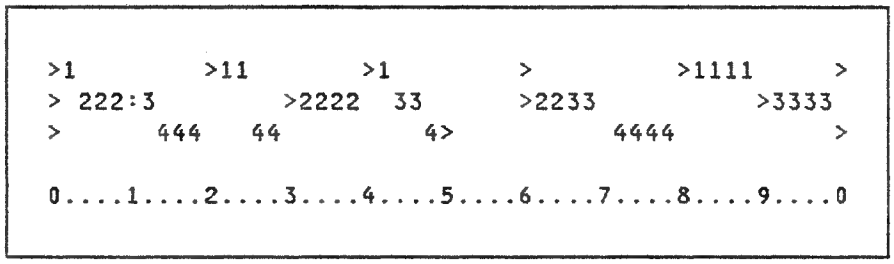

Figure 4.2
C-process Dispatching
Single CPU Example

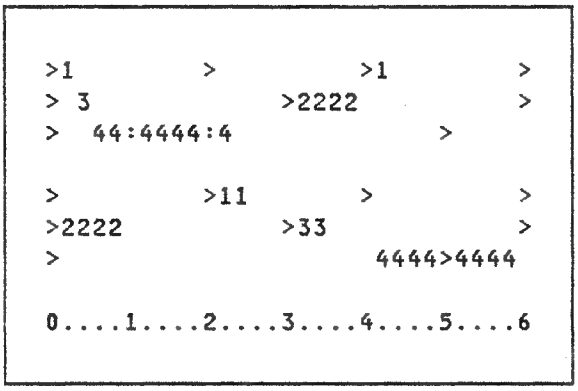

Figure 4.3
C-process Dispatching
Two CPU Example

R-processes is therefore defined that there are no explicit waiting conditions; an R-process is either running, or is ready to run as soon as a CPU can be switched back to it. The overall behavior of R-process dispatching is then as follows.

- In order for a process to be initiated as the result of occurrence of an event associated with a process source, a process model must be connected to the source using the CONNECT R-PROCESS MODEL instruction. If a process model is not connected then any occurrence of an event associated with the source is ignored.
- A source becomes **pending** if a process model is connected and an associated event occurs. It is removed from pending status by initiation of a process from the connected process model. If an associated event arises while a source is pending only a single pending status will result. A source is said to be **active** if at least one member of the family of the connected process model is in existence.
- An R-process model specifies whether or not more than one member of the family can exist at the same time. If a single-instance process model is

connected to a source, a pending condition for the source will not become effective while the source is active. A pending condition for a source connected to a multi-instance process model is immediately effective if the number of instances in existence is less than the number of CPU in the system; otherwise it becomes effective as soon as one of the processes is terminated.

- Initiation of a process for a pending source occurs as soon as the pending condition is effective and a CPU is available for assignment to the process. A CPU is available if it is idle. A CPU is made available by suspension of a runnning process if the running process is

  - a C-process, or

  - an R-process initiated by a source of lower priority than the pending source.

  Suspension occurs in the order just described, and in reverse priority order within the R-process set. Multiple pending sources are taken in priority order.

- Suspended processes are dispatched as soon as a CPU is available and there are no pending sources of higher priority. Dispatching is normally in reverse order of suspension. However, if a process is holding an access control gate, and if the gate has been requested by a higher priority process, a promotion technique is used to clear the gate. This technique is discussed in Section 5.10.

Execution of the instruction sequence of an R-process will continue, with or without periods of suspension, until the process terminates. Suspension is an internal condition which does not affect the functional behavior of a process. The only observable effect is that the execution time of the instruction sequence is longer than if suspension had not occurred.

4.7 Computation Cycle Overrun

Because all R-processes take precedence over all C-processes in assignment of CPU, the execution time of C-processes as observed by C-process dispatching includes all time the processes are suspended in favor of R-processes. Computation cycle scheduling therefore provides a measure of the total CPU utilization of the system, not just the part devoted to C-processes, so a computation cycle overrun indicates some degree of system overload.

Whether or not the overload requires remedial action depends on the processing environment and application mix. To service this dependence, a system overrun exception will be raised when an overrun condition is established. Data for the process model to be connected to the exception is specified at system initialization. In addition, as part of the specification of the computation cycle structure, two parameters can be adjusted to avoid unnecessary initiation of an overrun process.

The first parameter is a set of 8-bit integers, one for each computation cycle, called the **overrun sequence counts.** If a count is zero, an overrun for the associated cycle will be ignored. If a count is non-zero, it specifies the number of successive times an overrun for the associated cycle must be recorded before an overrun condition is established for that cycle.

Sequence counts provide a simple way

of masking bursts of R-process activity which only temporarily overload the system. Such burst will usually occur in all systems because of the stochastic distribution of events in time, but unless the average value of occurrences keeps activity high enough, the system is not really overloaded and the overrun indication will be damped out. Figure 4.4 illustrates this effect using the example of figure 4.2, with the addition of comma symbols to designate assignment of the CPU to some R-process.

In this example, process 1 is not assigned the CPU during the second period of computation cycle 1 and process 3 gets no CPU time for the first two periods of computation cycle 2, but by the end of basic cycle 12 the control flow will have returned to the basic pattern of figure 4.2. There is no need to invoke an overrun process unless one of the period multiples is an absolute time constraint.

The second parameter is the **overrun cycle indicator.** If this indicator is set equal to the identifier of some computation cycle, then an overrun of any cycle beyond the indicated one will be ignored, whether or not the overrun is established by sequence count. If the indicator is not equal to some identifier, all established overruns are effective. The cycle indicator provides a direct way of ignoring computation cycles whose time constraints are only nominal.

A system overrun condition is considered to be established whenever an overrun condition is established for a computation cycle within the range designated by the overrun cycle indicator. The scope of activity of the resultant overrun process is described in Section 9.4.

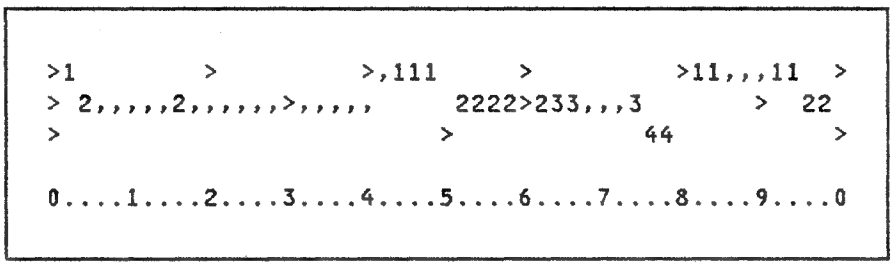

Figure 4.4
Single CPU Example
With R-processes Included

5.0 C-PROCESSES: GENERAL COMPUTATION

The behavior of a process is determined partly by the instructions available for execution, and partly by built-in functions of the system. The behavior patterns favored by the system functions and their associated conventions and protocols vary by process class, and for each class reflect a system view of the kind of activity appropriate to the class.

The view of C-processes as primarily data transformation activities is therefore carried beyond process dispatching, into initiation, structure, linkage, access to shared data, and exception handling. Moreover, as a data transformation is of little use without data, queuing facilities provide for explicit flow of data between processes, and as a means for the arrival of data to initiate a process.

5.1 Queues

A **queue** is an ordered collection of M-spaces, which may be an empty collection. M-spaces in a non-empty collection are referred to as **items** of the queue. Queues can be defined as part of the definition of a C-process model, or by the DEFINE QUEUE instruction (QDEF), and are made available in the following way.

- Each queue is assigned an arbitrary 32-bit **queue name** as part of its definition. Queue names must be chosen so as to completely distinguish one queue from another. However, reference to the queue in all active queuing instructions is by means of a **queue index** (**q.ix**), which is a 32-bit identifier assigned by the system. The QUEUE INDEX instruction (QIX) is used to determine the q.ix from the name.

- A queue defined by a DCPM instruction in association with a process model of the C-process class is called an **input queue** for the model, and any process of the process model family is called an **attendant** process of the queue. Items may be entered into into an input queue by any process irrespective of process class, but may be extracted from the queue only by an attendant process.

- More than one queue can be associated with a given process model, but a given input queue can be associated with only one process model. If a process model has multiple associated input queues, the queues are assigned a precedence sequence with respect to one another.

- An input queue serves as a process source for the associated process model: an item entered into the queue when the queue is empty automatically triggers a request for initiation of a process of the family.

- A queue defined by a QDEF instruction becomes a **public queue**, not associated with any process model. Items may be entered into a public queue by any process irrespective of class, and may be extracted by any process of the C-process class.

A queue, like a domain, is provided with a name in order that it have a permanent referent which can be assembled as a constant in storage or entered as input data. It is provided with a q.ix in order to optimize instruction execution performance. Except for the value zero, which always designates the

null queue, queue indices are model-dependent and not predictable. Consequently, a q.ix cannot safely be passed from one process to another. In order to assure correct results, any process not an attendant of a queue must execute a QIX instruction before referencing the queue.

Given a name, the QDEF instruction defines a queue of that name and returns its q.ix. If a queue with the given name already exists it is not redefined; the q.ix is returned with a condition code indicating prior existence. The QIX instruction, on the other hand, returns the q.ix of an existing queue corresponding to a given name, or else an indication that no such queue exists.

Sixteen bytes are withdrawn from the M-storage pool for every queue defined, to be used by the system for queue control. Queue definition will be rejected if M-storage for the control area is not available. The rejection is indicated by the condition code set for the DCPM or QDEF instruction attempting the definition.

5.2 Process Model Definition

The information required by a DEFINE C-PROCESS MODEL instruction (DCPM) is specified by means of a C-process Model Definition Block (CMDB). A CMDB must begin on a word boundary and have the format described in figure 5.1.

The DCPM instruction will define a new process model, replace an existing one, or delete a model entirely, depending on the content of the CMDB.

- A new model is defined if there is not already one with the name specified in field CMNME. If a model does exist and if deletion is allowed, it will be replaced or deleted; if deletion is not allowed, the definition attempt will be rejected. Replacement occurs if field CMINS is non-zero, deletion if the field is zero. Replacement consists of first deleting the existing model [Section 5.3], followed by definition of the new model. Deletion will not occur if the definition attempt is rejected for any reason.

- When definition is complete the CMDB area is immediately available for new use, as the process model control information is retained by the system in an area withdrawn from the B-storage pool. Definition will be rejected if storage is not available, and the rejection indicated by condition code return.

- The definition attempt will also be rejected if field CMCID does not identify a valid computation cycle, if any of the proposed input queue names duplicate the name of an input queue for another process model, if the proposed entry context space is not in custody of the defining process, or if M-storage is not available for queue control areas.

The remaining fields of the CMDB are not examined by the DCPM instruction. If they contain information which is invalid (e.g. field CMMOD does not contain a pointer), or later becomes invalid, the error will be noted at the time of attempted use, and an 'invalid process model' system exception will then be raised [Section 9.5].

The CMDB of an existing process model can be inspected at any time by execution of a STORE CMDB instruction (SCMDB) within any C-process or R-process. The data can be used to check field validity, to replace or

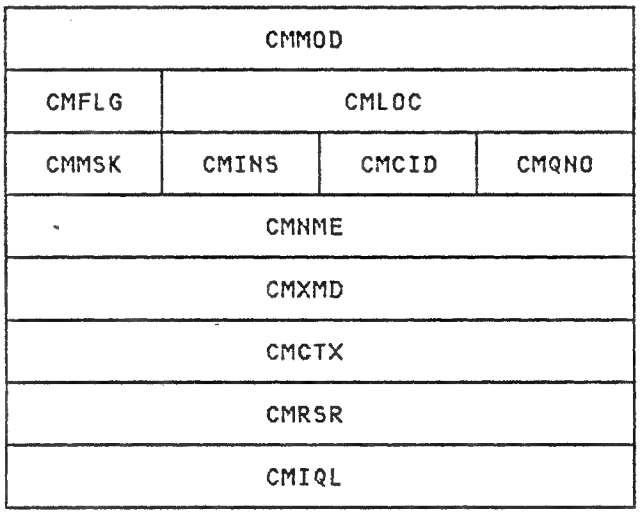

Figure 5.1
C-process Model Definition Block

Field	Offset	Bytes	Description and Use
CMMOD	0	4	A pointer to the module space which contains the initial instruction sequence to be executed by every process of the process family.
CMFLG	4	1	Bits defining conditions of usage for the process model and members of the family.

Bit	Value	Significance
0	0	Collect process statistics under control of collection instructions
	1	Do not collect process statistics
1	0	The initial value of the domain identifier for processes of the family is to be the identifier of the entry context space
	1	The initial value of the domain identifier is that of the common domain if a queue acted as source for the process, or the domain identifier of the source process if initiation is caused by an INIT instruction
2	0	The domain identifier may vary during execution
	1	The domain identifier is fixed during execution
3	0	The entry context space identified by field CMCTX is to be bound to the process model
	1	Do not change custody of the entry context space

4-6	-	Reserved
		7 0 Place this process model in system custody if custody is transferred
	1	Place in public custody on a transfer of custody

Field	Offset	Bytes	Description and Use
CMLOC	5	3	Base address within the module space identified by field CMMOD of the first instruction of the initial instruction sequence.
CMMSK	8	1	Value of the exception mask field for initial entry to processes of the family.
CMINS	9	1	An integer specifying the maximum number of family members in existence at the same time. If the value is between 1 and 254, the maximum is the value; if the value is 255, an unlimited number of instances is allowed. A value of zero indicates a deletion request.
CMCID	10	1	The identifier of the computation cycle with which the process model is to be associated.
CMQNO	11	1	An integer whose value specifies the number of input queues associated with the process model.
CMNME	12	4	The name of the model. Process model names are 32-bit permanent referents which must be unique within the set of process models, but need not differ from names in other name sets (e.g. queue names).
CMXMD	16	4	A pointer to the module space which contains the initial instruction sequence for handling process exceptions.
CMCTX	20	4	A pointer to an ordinary space which becomes the entry context for processes of the family.
CMRSR	24	4	Reserved for use as selection routine input data.
CMIQL	28	Var	Names of the input queues to be to be associated with the process model, listed in order of the precedence sequence for queue switching [Section 5.8]; the number of entries in the list is equal to the value of field CMQNO.

---

delete the model, or simply to determine the names of the input queues associated with the model.

5.3 Process Model Deletion

C-process models and queues are assigned to the custody of some process

family for protection against arbitrary deletion.

- A C-process model is assigned to the custody of the family of the defining process. It can be deleted by a DCPM instruction, operating in replacement or deletion mode, if the instruction is executed within a custodian process.

- An input queue is bound to the custody of its associated process model, and can be deleted only as part of the deletion of that model.

- A public queue is assigned to the custody of the family of the defining process. It can be deleted by a DELETE QUEUE instruction (QDEL) executed within a custodian process.

Unlike spaces, the continued existence of process models and queues is not tied to the continued existence of their custodian; they are transferred to alternative custody if their initial custodian is itself deleted.

- If bit 7 of field CMFLG of its CMDB is zero, a C-process model and its associated input queues are transferred to bound custody of the system. No process can then execute a valid DCPM instruction for the model, so it cannot be deleted or replaced except by re-initialization of the system.

- If bit 7 of the field is 1, the C-process model and its input queues are transferred to public custody. Any C-process or R-process can then execute a valid DCPM instruction for the model. If the DCPM causes the model to be replaced, the new model is transferred to family custody of the defining process.

- A public queue is always transferred to public custody.

When any queue is deleted, spaces resident in the queue are also deleted, the effect being as if a FREE instruction had been issued for each space. When a process model is deleted, spaces held in family custody are deleted, as is the entry context space if bit 3 of field CMFLG of the CMDB caused the space to be bound to the model. The input queues are then deleted, followed by deletion of the model. However, if deletion is an intermediate step in replacement of the model, objects bound to the model are not deleted if they are to be used for the new model.

- If field CMCTX of the new CMDB identifies the entry context space already bound to the model, the space remains in existence. If, at the same time, bit 3 of field CMFLG of the new CMDB indicates that the entry context space is not to be bound to the model, the space is placed into family custody of the replaced model.

- The queue list of the new CMDB is compared against the input queue list of the existing model and queues which appear in both lists are not deleted prior to replacement of the model.

The QDEL and DCPM instructions are synchronous to the process within which they are executed, so that deletion is effective at instruction completion. In some models of EPSILON systems, however, the M-storage and B-storage areas used for queue control and CMDB data may be retained by the system on instruction completion, for use in future queue and process model definition.

5.4 Initiation

A request for initiation of a C-process is triggered when either

- an INITIATE PROCESS instruction (INIT) is executed specifying a defined process model, or
- an item is placed into an input queue which was previously empty.

Items are entered into a queue by means of the ENQUEUE instruction (ENQ), whose operands denote the q.ix and the M-space. The INIT instruction, however, denotes the name of the process model, so that INIT is a direct request for initiation of a process of a particular family, while ENQ carries an implicit request for initiation of a process to serve a particular queue irrespective of associated process model family. Consequently, ENQ is made available to all processes, but INIT can be executed only within a process of the C-process class.

The sequence of action necessary to convert an initiation request into a C-process ready to be dispatched for the first time is carried out by microcode as a closed function of the system. The general principle embodied in the initiation microcode is to minimize the length of the initiation sequence without sacrificing its transparency. Therefore, even though no conditions are imposed on the residence of the initial instructions or process model control data, the initiation sequence chosen in any particular case will make use of residence conditions generated by previous activity in the system. The general procedure is as follows.

- When the request is triggered, a test is made to [illegible] if the process model control data is resident in M-storage. The data will be resident if a process of the family exists or is being terminated; it may still be resident if a process previously existed.
- If the data is in M-storage the membership status of the family is examined. If there are already as many processes of the family in existence as the maximum specified in field CMINS of the CMDB, the request is considered to be satisfied and no further action is taken. If the maximum has not been attained, the request becomes an active one. In either case, the INIT or ENQ instruction which triggered the request is then completed with condition code zero.
- If the data is not resident in M-storage, a search of B-storage is started. An ENQ instruction is completed when the search begins, as the existence of an input queue guarantees the existence of an associated process model; for the opposite reason, interpretation is suspended for an INIT instruction during the period of the search. The search is carried out in incremental steps at each entry to initiation service by dispatching [Section 4.2]. If the search is terminated by loading the control data into M-storage, the request becomes an active one. If there is no CMDB corresponding to the request, interpretation of the associated INIT instruction will resume, and it will be completed with a condition code indicating the request was dropped.
- Active requests are processed incrementally by initiation service [Section 4.2] until satisfied. If there is a process of the family which is in general wait condition, the request will

be satisfied by placing the process into ready condition; otherwise, an initial state vector will be prepared for a new process of the family. The state vector data resides in M-storage, so a new process may not be brought into existence until storage is available.

- When the state vector area is available, the CMDB fields CMMOD, CMLOC, CMXMD, and CMCTX are examined for validity. CMMOD must identify either the null space, or a module space for which field CMLOC designates an instruction location which lies within the space. CMXMD must identify either the null space or a module space. CMCTX must identify either the null space or an ordinary space. Initiation is suppressed with an 'invalid process model' system exception [Section 9.5] if any of the fields is discovered to be invalid.

- If field CMLOC identifies a module B-space, the space is examined to see if an M-space descendant generated by a linkage instruction exists. If one does not exist, it is formed by executing the equivalent of a CALL instruction.

- If field CMCTX identifies an ordinary B-space, and if there are no other processes of the family in existence, an M-space descendant of the space is formed by executing the equivalent of a LOAD instruction. The descendant, or the original space if field CMCTX identifies an M-space, becomes the **entry context** space for the process. Processes of the family can be supplied values, pointers, names, and other initial data by means of the entry context. If they are allowed write access to the space, it can also be used in conjunction with an access control gate [Section 5.10] to pass information between family members or generations.

- The initial state vector is then loaded with standard entry data and the new process placed in ready condition.

Because of the nature of the steps taken to initiate a process, the elapsed time between a request and the existence of a corresponding ready process cannot be predicted. It is possible to determine an upper bound to the time, but only for each EPSILON system individually, as the bound depends both on the model and the application mix. Once a C-process exists, however, its elapsed time behavior is governed by its own instruction execution during time apportioned within its computation cycle.

5.5 State Vector

The state vector of any process consists of the general registers, a Process Instruction Counter (PIC), and internal control data, all of which reside in M-storage. The internal control data is accessible only to the microcode, but the PIC can be obtained by the LOAD PROCESS INSTRUCTION COUNTER instruction (LPIC). The PIC is a double-word with the format described in figure 5.2.

The LPIC instruction loads the PIC of the process within which it is executed into a specified general register of that process. Field PCUR is loaded into the pointer register by executing the equivalent of an LP instruction; fields PFLG and LCUR are placed into the arithmetic register. LPIC is a modal instruction which can also be executed within an R-process

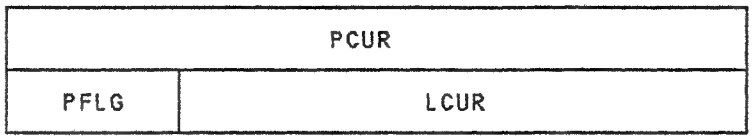

Figure 5.2
Process Instruction Counter

Field	Offset	Bytes	Description and Use
PCUR	0	4	Pointer to the M-space containing the next instruction to be executed
PFLG	4	1	Bits defining the condition and status of activity for the process

Bit	Value	Significance
0	0	normal condition for C-process or R-process; indicates D-process has no current space
	1	process activity is being monitored (C-process); the process has been unable to close an access control gate (R-process); there is an I/O request space current for the process (D-process)
1	0	normal
	1	the process has closed a data access gate
2	0	normal
	1	a process exception is not yet cleared
3	0	normal
	1	the process is being terminated
4,5	1-3	Length in halfwords of the last instruction executed
6,7	0-3	Current condition code

Field	Offset	Bytes	Description and Use
LCUR	5	3	Base address within the space identifed by field PCUR of the next instruction to be executed

---

or D-process. As shown in figure 5.2, the PIC for processes of all classes has the same format; content is also the same except for the interpretation of bit 1 of field PFLG [The function of the current space of a D-process is described in Section 7.6].

5.6 Process Entry

When a C-process is dispatched for

the first time its state vector is loaded with the following initial data.

- The PIC contains the location of the first instruction to be executed, as generated during the initiation sequence. Field LCUR contains the value of field CMLOC of the CMDB; field PCUR designates the space identified by field CMMOD, if it is an M-space, or an M-space descendant, if it is a B-space.

- Pointer register zero contains a pointer to the entry context space generated during the initiation sequence. The register is set up as if it had been loaded with an LP instruction. Arithmetic register zero contains zero.

- Pointer register 1 contains the null pointer. Arithmetic register 1 contains the q.ix of the input queue which triggered initiation, or zero if an INIT instruction was the trigger.

- All other general registers contain zero in the arithmetic register field and a null pointer in the pointer register field.

- The exception mask and domain identifier are set as specified in the process model.

If field PCUR of the PIC identifies a space to which the process does not have read access, the process is terminated immediately and an invalid process model system exception is raised. If the space is the null space, instruction execution is not started. All spaces in the input queue which triggered initiation, if any, are deleted from the system, and termination is requested as if an EXIT instruction with an operand value of zero had been executed [Section 5.12]. If the space is not null, instruction execution begins with the first instruction and continues until a CPU switch occurs, at which time interpretation of the instruction sequence is suspended. If the switch was caused by expiration of a basic cycle, the state vector will be preserved intact, and when the process is next dispatched instruction execution will resume from the point of suspension. CPU switching due to computation cycle activity is therefore invisible to all processes. There are, however, two instructions by which a process can explicitly return control of its assigned CPU to dispatching.

- The IDLE instruction is used to indicate that the process activity for this computation cycle is finished. The process is suspended in ready condition at instruction completion. It will be dispatched during the next computation cycle with the state vector preserved, except that the process instruction counter is set to the reentry location specified by the operand of the IDLE instruction.

- The WAIT instruction is used to indicate that the process must wait for some event or occurrence. The process is suspended in general wait condition at instruction completion. It will not be dispatched again until the wait is removed by an initiation request, or by expiration of the maximum wait time specified by the operand of the WAIT instruction. The state vector is preserved, so that when the process is next dispatched the first instruction executed is the one following the WAIT.

Control of the CPU will also be returned by a termination request, but the process will not then again be

dispatched.

5.7 Linkage

Normal system procedure is to execute instructions drawn sequentially from the same module M-space. After an instruction is fetched from the location specified by the process instruction counter, the address in field LCUR is incremented by the number of bytes of the instruction to yield the location of the next instruction. If the address becomes larger than the extent of the module space, it is reduced by the extent, so that instruction fetch wraps around the space.

The normal procedure is changed by branching and linkage instructions. Branching instructions, such as BRANCH ON CONDITION or BRANCH AND LINK, change the fetch sequence while leaving unchanged the space from which they are fetched. The branch address generated by the instruction replaces the the address in field LCUR of the PIC, but no action is taken to change the space identified by field PCUR. The linkage instructions provide for a change of space as well, in order to allow instructions from more than one module space to be executed within the same process.

The CALL instruction, which is a modal instruction executable within all processes, links to a new space. When executed within a C-process or D-process:

- The operand can identify either a module B-space or M-space for which the process has read access. If a B-space is identified it is examined to see if an M-space descendant generated by a previous CALL exists. If one does not, it is formed by executing the equivalent of a LOAD instruction. The process is placed into I/O wait dispatching condition until the loading is completed.

- When the M-space is available, either as the original operand or as the descendant of a B-space, the PIC is saved in a linkage control area of the state vector. Field LCUR is then set to the base address specified by the instruction operand, field PCUR is set to identify the new M-space, and the instruction is completed.

When executed within an R-process the operand of a CALL must identify an M-space or else a specification exception will occur; the instruction behavior is the same in all other respects.

An M-space allocated by linkage as a descendant of a B-space, either during initiation or by a CALL instruction, is assigned to the private custody of the associated process, but its custody flag is not turned on. It will be deleted automatically by the system when no longer in use, or when the process is terminated. However, as a consequence of the behavior of the linkage mechanism, the space may become the source of instructions for other processes executing concurrently. The reference count of the space is therefore incremented by the CALL instruction, so that deletion will be delayed, if necessary, until the space is not in use by any process.

Similarly, the M-space from which instructions were fetched prior to a CALL continues to exist, and will continue to exist irrespective of whether it is in use by other processes, until released by action of the process within which the CALL was executed. Spaces are released either by termination of the process or by execution of a RETURN instruction.

RETURN reverses the CALL procedure, releasing the current space as the source of instructions for the process and restoring the previous space.

- The reference count of the space which was released is decremented, so that if it is not in use by any other process it will be deleted from the system.

- The PIC is set to the value saved from the last CALL instruction executed by the process, so that normally the next instruction fetched will be the one following that CALL. If no previous CALL was executed, the RETURN is treated as an EXIT instruction [Section 5.12].

- The operand of RETURN specifies instruction length and condition code bits in the format of field PFLG of the PIC. After the PIC is restored to its previous value, field LCUR is incremented by the number of half-words indicated by the length code, and the instruction is completed by setting the condition code to the value specified by the operand.

CALL and RETURN are paired in the manner of stack operations, with CALL corresponding to the push and RETURN to the pop. The pairing is strictly maintained, as there is no instruction which will do a return linkage to a CALL prior to the latest one. A process is not, however, required to execute as many RETURN instructions as CALL instructions.

5.8 Communication Via Queues

Apart from changing the PIC and the condition code, the linkage instructions do not affect the process state vector. Hence, data or data locations can be exchanged between different instruction sequences of a single process by means of the general registers. However, exchange of information between different processes requires another mechanism, as the registers of a process are private to that process.

Input queues are the principal means of exchanging information between C-processes. They are also the only means by which R-processes and D-processes can transmit data to be acted upon by C-processes. The data to be exchanged or transmitted is placed into an ordinary M-space and an ENQUEUE instruction (ENQ) is executed specifying the space and the q.ix. An attempt to enqueue a module M-space or a B-space of any kind will be rejected with a specification exception.

Because a queue is specified rather than a process, communication is indirect; the enqueuing process, in fact, cannot even determine the process model family associated with the queue. But since ENQ carries an implicit request to initiate an attendant of the queue [Section 5.4], a permanent basis of communication can be established by assignment of each queue as the input mechanism for one or more data transformation functions.

The items in a queue are ordered by their sequence of entry, with the least recently entered item being the **top** or **head** item, the most recently entered the **bottom** or **tail** item. Successful execution of an ENQ instruction enlarges the designated queue so that the referenced M-space is the new bottom item, with the previous bottom item becoming its predecessor. An M-space entered into a queue becomes part of the queue. It loses addressability as a space, and passes out of the custody of the enqueuing process or process family into bound custody of the queue, where it cannot be deleted or reclas-

sified by any process. Any attempt by a process to load a stored pointer to the space into a pointer register while the space is in the queue will return a 'space not available' condition code.

In a manner similar to the action of the FREE instruction, and for the same reasons [Section 3.4], entry of an M-space into a queue is delayed until all processes which have gained access to the space no longer need to reference it. The ENQ instruction substitutes a null pointer for any pointer to the space in all pointer registers of the process executing the ENQ, and the reference count is decremented for each substitution. If the count is not then zero, the space is held in abeyance until the count becomes zero, at which time its custody flag is turned off and it will be placed into the queue. The ENQ instruction, however, always appears synchronous to a process.

Once an attendant process is initiated, it may use the DEQUEUE instruction (DEQ) to receive the queued data items. The position and provenance of the item extracted from a queue by DEQ can be specified by instruction options to be that item

- nearest to the top of the queue, or nearest to the bottom of the queue, which
- has any domain identifier, or has the domain identifier currently assigned to the process within which the instruction is being executed.

A pointer to the extracted item is placed into the pointer register designated by the DEQ instruction, and the item returns to normal M-space existence. Custody is transferred to the dequeuing process or its family, according to the value specified by the custody option, with the custody flag turned on, and with the reference count set to the value 1. Except for its content and domain identifier, which remain unchanged, the space then cannot be distinguished from one obtained by the process using an ALLOC instruction.

If the queue is empty, the DEQ instruction returns a null pointer and a condition code indication. The process may ignore the empty condition, request termination, or switch to another input queue, if there is one. It may also suspend activity with a QUEUE WAIT instruction (QWAIT), which puts the process into the queue wait dispatching condition. Unlike general wait, which is cleared by any initiation trigger, queue wait is cleared by entry of an item into the specific queue designated by the instruction. The process is dispatched again only when that occurs, with the instruction following the QWAIT as the next instruction to be executed.

Although any input queue associated with the family can be accessed by a DEQ instruction, the system recognizes one of the queues as the **current queue** of the process and will dequeue from it unless the instruction specifies otherwise. The current queue is set initially as the one which triggered initiation of the process, or as the null queue if an INIT instruction was the trigger; its q.ix is placed into arithmetic register 1 at first entry to the process [Section 5.6]. The QUEUE SWITCH instruction (QSWCH) changes the setting of the current queue

- to the queue which is higher in the precedence sequence than all other non-empty queues, or
- to the null queue if all queues are empty.

The q.ix of the new current queue is

returned in the arithmetic register designated by the instruction.

The null queue may be specified in any instruction which takes a q.ix as an operand. It is treated as a queue which does not exist or is always empty, as appropriate for the instruction:

- for ENQ the result is release of the space as if a FREE had been executed
- for QWAIT the wait condition is not generated and the instruction becomes equivalent to an IDLE
- for DEQ the result is the 'queue empty' condition code return

The null queue is useful in writing a module to execute without change within processes intiated by INIT as well as processes intiated by queuing of data.

5.9 Domain Identification

The DEQ instruction is the only instruction which can cause a change of domain identifier for a C-process. This leads to the following simple rules for assignment of domain identifier to C-processes.

- If bit 1 of field CMFLG of the CMDB is zero, the initial domain identifier is the domain identifier of the entry context space. An assignment can always be made, as the null space belongs to the common domain.
- If bit 1 of field CMFLG is 1, and if an input queue triggered initiation, the common domain identifier is assigned to the process; if an INIT instruction triggered initiation, the initial identifier is the one which the initiating process had when the INIT was executed.
- If bit 2 of field CMFLG is 1 the intial identifier is retained by the process throughout its lifetime. If bit 2 is zero, the domain identifier is changed by execution of a DEQ instruction to the domain identifier of the space just dequeued; the new domain identifier remains current until the next dequeue.

The current domain identifier of a process becomes the domain identifier of any ordinary space allocated by the process using an ALLOC instruction [Section 3.5].

5.10 Shared Data

Processes can always exchange information by simply addressing data directly at a common location. If the data is constant or can be updated in place by some instruction, any group of processes can share the data as long as they have access to the space of residence. If data update requires more than one instruction, direct access will usually obtain invalid data because processes in EPSILON systems behave as if they all are advancing concurrently. Some additional mechanism is required, then, if complex data is to be shared.

The mechanism provided, called an **access control gate**, or simply a **gate**, achieves the desired result by assuring that the advancement of a group of processes will be mutually exclusive in time. A gate is any word, located on a word boundary in some M-space, which becomes a special resource because of reference by CLOSE GATE (CLOSE) and OPEN GATE (OPEN) instructions. A gate belongs to a process which executes a successful CLOSE instruction; it is released by a successful OPEN instruction. A gate which belongs to some process is

said to be **closed**, otherwise it is **open**.

A CLOSE followed eventually by an OPEN referencing the same gate and executed by the same process delimits the instruction sequence included between the matched pair of gating instructions as an exclusive sequence. A process is guaranteed that an instruction sequence delimited by a matching pair of gating instructions will be executed exclusively in time relative to any instruction sequence of any other process of the same process class which is delimited by a pair of gating instructions referencing the same gate. Exclusivity is limited to processes of the same process class in order to be consistent with normal dispatching activity, and is not extended to D-processes.

To operate properly, a gate must be initialized to a zero value and then referenced only by CLOSE and OPEN instructions. To use a gate for sharing data, a group of C-processes or a group of R-processes need only associate the gate with the data and act as follows:

- the gate is assembled to have a zero value or is cleared to zero by one of the processes before use of the first gating instruction
- a CLOSE is executed by any process prior to accessing the data
- an OPEN is executed by a process as soon as it is through with the data.

A gate need not be restricted in use to one process class, though it is good practice to do so. If a gate is open, it can be closed by any C-process or R-process. Once closed, the gate and gating instructions referring to it behave modally, as the connection between gating and dispatching is established when a process attempts to close a gate which is already closed. If a gate belongs to a C-process:

- An attempt to close the gate by an R-process will be rejected with a condition code indicating incorrect process class.
- An attempt to close the gate by another C-process will put that process into gate wait dispatching condition, and an internal gate queue will be formed. As long as the gate is closed any other process which attempts to close it will be placed at the end of the queue.
- If there is no gate queue, an OPEN instruction simply returns the gate to open status. If there is a gate queue, then the OPEN has the effect of completing the CLOSE instruction for the process which is at the head of the queue. Thus, when the OPEN is completed
  - the gate belongs to the process which was at the head of the gate queue
  - that process has been removed from the queue and is in ready condition.

  The process within which the OPEN was executed is no longer involved with the gate.

The dispatching treatment accorded a C-process in gate wait is determined by action of the selection routine which services the computation cycle of the process. The three standard selection routines assign the CPU which would have been assigned to a process in gate wait to the process which owns the gate, thus giving that process the CPU time of all processes

in the gate queue. This form of dispatching promotion has the effect of unstacking a gate queue more rapidly the longer it becomes.

If a gate belongs to an R-process:

- An attempt to close the gate by a C-process will be rejected with a condition code indicating incorrect process class.
- An attempt to close the gate by another R-process will result in invocation of dispatching to suspend the process and switch to another. At the same time, the process to which the gate belongs is temporarily promoted to a dispatching priority higher than the process just suspended. As long as the gate is closed, the process to which it belongs continues to be promoted to a priority above that of any process which attempts to close the gate.
- If a process has not been promoted, an OPEN instruction simply returns the gate to open status. If the process executing the OPEN has been promoted, the OPEN returns it to its original priority and causes a process switch. Normal dispatching activity will then assure that the highest priority suspended process is assigned a CPU, and the CLOSE instruction which caused the suspension is completed when the process starts running.

Although dispatching promotion techniques have a substantial effect on the behavior of the system, individual processes need not take them into account as they do not affect the functional behavior of the CLOSE and OPEN instructions.

5.11 Deadlock Avoidance

The OPEN instruction acts like a process switch in forcing results of previous instructions to be brought into physical and conceptual agreement [Section 2.9]. Consequently, gating is a general means of serializing process execution, which can be used for process synchronization as well as for resolving resource usage conflicts. However, as a process may close more than one gate at a time, it is possible to create a deadlock condition where two or more processes block each other's advancement. For example, the following two processes will very likely create a deadlock:

Process1	Process2
CLOSE G1	CLOSE G2
CLOSE G2	CLOSE G1
.	.
.	.
OPEN G2	OPEN G1
OPEN G1	OPEN G2

Deadlocks can be forestalled by use of known avoidance techniques. One such technique is to assign a prescribed sequence to a set of gates, and to have any process which is to close one of the gates first close all gates preceding it in the sequence. Thus, the following processes can never create a deadlock:

Process1	Process2	Process3
CLOSE G1	CLOSE G1	CLOSE G1
CLOSE G2	.	CLOSE G2
CLOSE G3	.	.
.	.	.

The system recognizes some potential deadlock conditions and will take avoidance action or will alert a process so it may take action.

- An attempt by a process to close a gate it has already closed or to open a gate it has not previously closed will be rejected with a condition code indicating the rejection.

- An attempt to execute a CLOSE or OPEN using a word whose content is not consistent with the content of a gate will be rejected with a condition code indicating the inconsistency. As an aid in the preservation of gates from inadvertant destruction, CLOSE and OPEN are treated as read instructions, even though they alter the gate contents. Consequently, a gate can be located in a space to which the processes which reference it have only read access.

- A C-process will not be placed into a gate queue if it is already holding a gate; the CLOSE instruction is rejected with a condition code indicating the gate is not avilable to the process. Recovery action is up to the process, as the rejection only warns against a possibility that may never occur.

- A gate count is maintained for each M-space, which is incremented when a CLOSE is executed referring to a gate located in the space and decremented when an OPEN is executed. Any request for deletion of the space will be held in abeyance until the count becomes zero.

- Deletion of the state vector of a terminated process is delayed until all gates which belong to the process have been opened [Section 5.12].

If a deadlock does occur for a group of C-processes, processes not involved in the deadlock will continue to advance normally. The deadlocked processes are usually not recoverable. The deadlock may not even be detected, except indirectly by system behavior. An R-process deadlock, however, affects all other processes, so provision is made for detecting and breaking R-process deadlocks [Section 6.5].

5.12 Termination

A request for termination of a process is triggered when either

- an EXIT instruction is executed by the process, or

- a TERMINATE PROCESS instruction (TERM) is executed specifying the process model.

EXIT can be executed within any process to request its own termination. The request can signify normal completion, or recognition of an exception condition which precludes continuation of the process. TERM requests termination of all members of a process model family currently active. The request is accepted if executed within any process which is allowed to delete the process model, otherwise it is rejected with a condition code indication.

EXIT and TERM are synchronous to the process within which they are executed; termination is effective at instruction completion in the sense that the process to be terminated will not be dispatched again. Actual deletion of the process requires recovery of resources still held by the process, and is carried out asynchronously by microcode as a closed function of the system. For C-processes the steps necessary for deletion are processed incrementally by C-process termination service.

- A termination request sets bit 3 of field PFLG of the PIC to 1. When this bit is on, dispatching will not assign a CPU to the process when it is selected, but will substitute termination service in its place.

- If the process has one or more

I/O requests outstanding, termination service simply executes the equivalent of an IDLE. Idling will continue each time the process is selected until I/O activity for the process has ceased.

- When there is no more I/O activity, termination service deletes the spaces which remain in private custody of the process. The equivalent of one FREE is executed at each entry to the process until all spaces have been released.

- When all spaces have been deleted, the linkage control area of the state vector is tested for uncompleted linkage sequences. The equivalent of one RETURN is executed at each entry to the process until the linkage stack vanishes.

- At its next entry, termination service tests whether the process is holding any access control gates, or is waiting in a gate queue. If not, the state vector is deleted from M-storage and the process disappears from the system.

- If the process is holding a gate or is in a gate queue, it will be removed from its computation cycle but the state vector will remain in M-storage. Each time after that a CLOSE is executed referencing a gate which belongs to the process, an OPEN will be inserted in front of it. Eventually all gates belonging to the process will be opened; the state vector will then be deleted from M-storage.

For purposes of initiation, a terminating process does not count against the maximum number of processes allowed the family. If the maximum has actually been attained, an initiation request received while termination is in progress will be held in abeyance; it will become active when the process is removed from the computation cycle.

When the process terminated is the only one of its family in existence and no initiation request has been received, the input queues associated with the process model are primed to trigger initiation requests at first entry of a space irrespective of whether or not they are empty. A process is therefore not required to empty its input queues in order to assure continued initiation of processes of the family.

5.13 Exception Handling

The 15 process exceptions are assigned exception codes and divided into four severity classes, as shown in figure 5.3. The treatment of a process exception depends upon whether or not a non-null exception module is currently defined for the process. An exception module specified by a CMDB becomes current for all processes of the family as soon as they are initiated. A DEFINE EXCEPTION MODULE instruction (DXM) can then be executed within a C-process to define a new current exception module. DXM also returns a pointer to the old exception module, as an aid to those applications which may wish to change exception handling with linkage.

If a process exception which is not maskable, or is not masked, occurs within a C-process for which a non-null exception module is current, it is treated as follows.

- If the exception module pointer designates a space no longer in existence, the current exception module for the process is changed

to the null module, and an invalid process model system exception is raised [Section 9.5]. If the space exists, a linkage is generated to it exactly as if a CALL instruction had been inserted between the instruction which caused the exception and its successor.

- The PIC in effect at the time of the exception is stored in an **exception record** to which the process is given read access. An exception record is 32 bytes in extent and also holds the contents of general registers zero and 1, in the format described in figure 5.4.

- The state vector at entry to the exception handling sequence is set so that

  - the new PIC reflects the simulated CALL, with field PCUR containing a pointer to the exception module or an M-space descendant of it, with bit 2 of field PFLG set to 1 and bit zero set to zero, and with field LCUR containing zero as the entry base address; the condition code is not changed except in the case of forced exceptions

  - pointer register zero contains a null pointer

  - arithmetic register zero contains the exception code in the form of a positive fixed-point integer

  - general register 1 contains the location of the exception record

  General registers 2 through 15 remain as they were prior to the exception, as does all other state vector data which can be directly altered by the process (e.g. the exception mask).

The exception handling routine can take whatever action deemed desirable to recover from the exception, and then will conclude with a RETURN, indicating recovery has occurred, or a termination request, indicating continuation of the process is useless. If a RETURN is executed, the exception routine must set the state vector to the values desired at resumption of the normal instruction sequence. For this purpose, the exception mask can be set by the SET EXCEPTION MASK instruction (SXM), and general registers zero and 1 can be restored from the exception record. The return should take into account the time at which the state vector data was stored.

- An instruction which causes a class 1 or class 2 exception is suppressed before the process state vector is altered or data in storage is modified.

- Class 3 and class 4 exceptions are raised after the instruction is terminated or completed. For these exceptions the PIC location has been updated and registers or storage may be modified.

Consequently, if an exception routine wishes to bypass retry of an instruction which caused a class 1 or class 2 exception, the corresponding instruction length should be inserted into the operand field of the RETURN instruction.

If a process exception occurs when a null exception module is current, system microcode will take the following standard recovery action:

- class 1 and class 2 exceptions cause an EXIT instruction with operand value zero

Class	Exception	Code
1	Operation	1
	Execute	2
	Access	3
2	Addressing	4
	Specification	5
	Data	6
3	Forced	7
4	Fixed-point overflow	8
	Fixed-point divide	9
	Decimal overflow	10
	Decimal divide	11
	Exponent overflow	12
	Exponent underflow	13
	Significance	14
	Floating-point divide	15

Figure 5.3
Process Exceptions by Class

LPIC
PR0
PR1
AR0
AR1
FRCE

Figure 5.4
Exception Record

Field	Offset	Bytes	Description and Use
LPIC	0	8	Content of the PIC in use when the exception was raised. The value is identical to that stored in the linkage control area by the CALL simulated to enter the exception routine.

PR0	8	4	Content of pointer register zero when the exception was raised.
PR1	12	4	Content of pointer register 1 when the exception was raised.
AR0	16	4	Content of arithmetic register zero when the exception was raised.
AR1	20	4	Content of arithmetic register 1 when the exception was raised.
FRCE	24	8	Record extension containing data pertinent to a forced exception [Section 5.14]. The field has no significance for other exceptions.

- a class 3 exception is recorded if process statistics are being accumulated; the process then resumes at the instruction following the one which caused the exception.
- class 4 exceptions are ignored.

An exception routine can execute any instruction available to a C-process, including DXM. However, an exception which occurs while an exception routine is the current activity of a process (i.e. while bit 2 of field PFLG of the PIC is set on) is treated as if a null exception module were current.

5.14 Forced Exceptions

A process exception can be forced when a monitor trace record is stored for a process. Tracing is one of the process monitoring mechanisms of EPSILON systems which actively probe process activity. The action of these mechanisms in relation to a process is governed by bit zero of field PFLG of the PIC. A process will be actively monitored only when the bit is turned on; if the bit is zero, process activity is not probed, and no monitor exceptions will be forced. The value of the monitor bit in the PIC is controlled by the SET MONITOR MASKS instruction, which also specifies what is to be monitored or traced. This instruction is described in Section 10.8, together with a description of the process monitoring mechanisms.

A process exception can also be forced by execution of a FORCE PROCESS EXCEPTION instruction (FPX) within any process. The FPX instruction is a passive monitoring mechanism whose action is not governed by the monitor bit of the PIC, but by a separate eight-bit field in the state vector, called the **breakpoint mask**. When an FPX is encountered, each bit of the current breakpoint mask is combined with the corresponding bit of the mask field of the instruction by a logical AND operation. If the result is non-zero, an exception is forced; if the result is zero, process activity continues with the instruction following the FPX. The breakpoint mask of a process is set to zero at initial entry. It can be changed by execution of a SET BREAKPOINT MASK instruction (SBKM).

An exception record stored as the result of an FPX contains data in the FRCE field [Figure 5.4] in the format described in figure 5.5. A monitor

trace record has the same format for its FRCE field, but the internal fields contain quite different data [Figure 10.14]. The records are distinguished from one another by the condition code set upon entry to the exception module. If the condition code is zero, the exception was forced by an FPX instruction. If the condition code is 1, the exception is due to process monitoring.

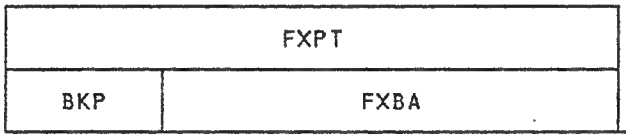

Figure 5.5
Extension of Exception Record
Stored by FPX Instruction

Field	Offset	Bytes	Description and Use
FXPT	0	4	Pointer portion of the location generated by the operand of the FPX instruction.
BKP	4	1	Breakpoint conditions which caused the interrupt to be forced.
FXBA	5	3	Base address portion of the location generated by the operand of the FPX instruction.

5.15 Instruction Descriptions

DEFINE QUEUE

QDEF R1,D2(X2,B2) <RX>

The queue list is scanned for a queue whose name is the same as the contents of the word located by the second operand. If such a queue is found, its q.ix is placed into arithmetic register R1 and the instruction is completed with condition code 1.

If no such queue exists, a queue is defined having the given name. The queue is made a public queue assigned to the custody of the family of the process within which the instruction is being executed. The q.ix of the new queue is placed into arithmetic register R1 and the instruction completed with condition code zero.

Process Class: C

Condition Code:

0	Queue defined
1	Queue already exists
2	-
3	-

Exceptions: None

QUEUE INDEX

QIX R1,D2(X2,B2) <RX>

The queue list is scanned for a queue whose name is the same as the contents of the word located by the second operand. If such a queue is found, its q.ix is placed into arithmetic register R1 and the instruction is completed with condition code zero. If no such queue exists, the register is cleared to zero and the instruction completed with condition code 1.

Process Class: C

Condition Code:

0	Queue exists
1	Queue does not exist
2	-
3	-

Exceptions: None

DEFINE C-PROCESS MODEL

DCPM D1(B1) <SI>

The instruction is suppressed with a specification exception if the operand does not define a location on a word boundary. If the location is valid, the process model list is scanned for a C-process model whose name matches the name stored in field CMNME of the CMDB located by the operand. If no such model exists, the computation cycle identifier in field CMCID of the proposed CMDB is tested for validity. If it is not valid the instruction is terminated with condition code 1.

Field CMQNO is examined for input queue count. If non-zero, the proposed names are compared against the the queue list, and if any name duplicates that of an existing queue the instruction is terminated with condition code 1. If the queue names are not duplicates, field CMCTX is examined for an entry context space pointer. If the pointer is non-null and does not identify an ordinary space in private or family custody of the process within which the instruction is being executed, the instruction is terminated with condition code 1. An attempt is then made to obtain B-storage for the CMDB data and

M-storage for queue control areas. The instruction is terminated with condition code 3 if storage is not available.

If storage is available, the input queues are added to the queue list, the CMDB information stored in the B-storage area, and the process model assigned to the custody of the family of the process within which the instruction is being executed. If field CMCTX is non-null, and if bit 3 of field CMFLG is 1, the entry context space is removed from its current custody and bound to the custody of the process model.. The instruction is then completed with condition code zero.

If a process model exists which matches the name in the proposed CMDB it is tested for deletion status. Deletion is allowed if the model is in custody of the family of the process within which the instruction is being executed or if the model is in public custody. The instruction is terminated with condition code 2 if deletion is not allowed.

If field CMINS of the proposed CMDB is zero, the model and its input queues are deleted from the system. The entry context space is deleted if in bound custody of the modal. A queue is deleted by releasing any spaces in the queue, followed by removing the control area from the queue list and returning it to the M-storage pool. The model is deleted by terminating all existing members of its family, followed by returning the CMDB control area to the B-storage pool. The instruction is then completed with condition code zero.

If field CMINS of the proposed CMDB is non-zero, the list of proposed queues is compared against the the input queue list of the existing model. Names that match are removed from the proposed list and saved separately. An attempt is then made to obtain M-storage for queue control areas for the remaining queues, if any. The instruction is terminated with condition code 3 if storage is not available.

If storage is available, the existing model and the non-matching input queues are deleted from the system. The proposed entry context space is compared to the entry context space of the existing model. If the spaces are not the same, and if the space of the existing model is bound to its custody, that space is deleted from the system. The proposed space then becomes the entry context space; whatever its previous custody, it is bound to the custody of the model if bit 3 of field CMFLG is 1, and placed into family custody of the model if bit 3 is zero.

The new input queues are added to the queue list, the CMDB information stored in the B-storage area of the previous model, the previously existing queues associated with the new model, and the new model is assigned to the custody of the family of the process within which the instruction is being executed. The instruction is then completed with condition code zero.

Process Class: C

Condition Code:

0	Model defined or deleted
1	Invalid CMDB format
2	Deletion not allowed
3	Storage not available

Exceptions:

Specification

STORE CMDB

SCMDB R1,D2(X2,B2) <RX>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. If the location is valid, the process model list is scanned for a C-process model whose name matches the name contained in arithmetic register R1. If no such name can be found, the instruction is terminated with condition code 2.

If a model exists with the specified name, its CMDB data is stored into successive word locations starting at the location defined by the second operand, up to the number of words required to store the complete CMDB. If storing a word would require exceeding the space boundary, the instruction is terminated at that point with condition code 1. It is completed with condition code zero if the full CMDB is stored.

Process Class: C

Condition Code:

0	Full CMDB stored
1	Partial CMDB stored
2	Process model not found
3	-

Exceptions:
Specification

DELETE QUEUE

QDEL R1 <RR>

If the contents of arithmetic register R1 are not consistent with a q.ix the instruction is suppressed with a specification exception. If the queue identified by the q.ix is not a public queue the instruction is terminated with condition code 1. If the public queue is not in private or family custody of the process within which the the instruction is being executed, or not in public custody, the instruction is terminated with condition code 2. Otherwise the queue is deleted from the system, arithmetic register 1 is cleared to zero, and the instruction is completed with condition code zero.

Process Class: C

Condition Code:
0 Queue deleted
1 Queue not public
2 Deletion not allowed
3 -

Exceptions:
Specification

INITIATE PROCESS

INIT D1(B1) <SI>

The process model list is scanned for a C-process model whose name matches the name stored in the word located by the operand. The instruction is terminated with condition code 1 if no such model can be found. If a model exists the request is accepted and the instruction completed with condition code zero. Initiation will be carried out by initiation service of dispatching [Section 5.4].

Process Class: C

Condition Code:
0 Request accepted
1 Process model not found
2 -
3 -

Exceptions: None

LOAD PROCESS INSTRUCTION COUNTER

LPIC R2 <RR>

The PIC of the process within which the instruction is being executed is loaded into general register R2. Field PCUR is loaded into the pointer register with the same effect as if loaded with an LP instruction [q.v. Section 3.7] except that the condition code is not set to reflect pointer availability, and the combined content of fields PFLG and LCUR is placed into the arithmetic register.

The instruction is then completed by setting the condition code to indicate the type of PIC stored.

Process Class: C,R,D
Modal

Condition Code:
0 C-process class
1 R-process class
2 D-process class
3 -

Exceptions: None

IDLE

IDLE M1,R2 <RR>

The contents of arithmetic register R2 replace the base address in field LCUR of the PIC, and the value of bits 2 and 3 of mask field M1 replace the condition code in field PFLG. Control of the CPU assigned to the process is then returned to dispatching. The process will begin execution at the new location with the new condition code when next dispatched.

Process Class: C

Condition Code: Set by instruction operand

Exceptions: None

WAIT

WAIT R1,R2 <RR>

The contents of arithmetic register R2 replace the base address in field LCUR of the PIC, the process is placed into general wait dispatching condition, and control of the CPU assigned to the process is returned to dispatching.

The contents of arithmetic register R1 are interpreted as a 32-bit positive integer specifying the maximum number of computation cycles the wait is to be in effect. The count is decremented by 1 for each cycle following the instruction, and the process will be placed into ready condition when the count goes to zero if an initiation request has not occurred prior to that time. When next dispatched, the process will begin execution at the new instruction location with register R1 containing the count current when the process was placed into ready condition.

Process Class: C

Condition Code: Unchanged

Exceptions: None

CALL

CALL M1,R2 <RR>

If the space identified by pointer register R2 is not a module space the instruction is suppressed with a data exception. It is suppressed with an access exception if the process does not have read access to the space. If the space is a B-space and the instruction is being executed within an R-process, the instruction is suppressed with a specification exception.

If the space is a B-space it is tested for an existing M-space descendant. If none exists, the C-process is placed into I/O wait dispatching condition and the equivalent of a LOAD instruction is inserted in front of the CALL. The I/O wait is removed after the loading is complete and the CALL interpretation restored. The new M-space is placed in the private custody of the process within which the instruction is being executed.

The process instruction counter is saved in the linkage control area of the state vector. The contents of general register R2 then replace the instruction counter, bits 2 and 3 of mask field M1 replace the condition code, and the instruction is completed by incrementing the reference count of the space identified in the instruction counter by 1.

Instruction execution for the process resumes at the new location with the new condition code.

Process Class: C,R,D
Modal

Condition Code: Set by instruction operand

Exceptions:
Access
Specification
Data

RETURN

RETURN M2 <RR>

The linkage control area is examined for a previous CALL executed within the process. If none exists the instruction is interpreted as an EXIT instruction.

If a previous CALL was executed, and if the process has private custody of the M-space from which the RETURN was fetched, the space is released from custody of the process. The reference count of the space is decremented by 1, and if the count becomes zero a deletion request is made for the space.

The PIC saved from the last CALL executed is cleared from the linkage control area and becomes the new PIC of the process. Field LCUR of the PIC is then incremented by the number of half-words specified by the value of the field consisting of bits zero and 1 of the operand M2, bits 2 and 3 of the operand replace the condition code, and the instruction is completed.

Instruction execution for the process resumes at the instruction specified by the updated PIC, with the new condition code.

Process Class: C,R,D

Condition Code: Set by instruction operand

Exceptions: None

ENQUEUE

ENQ R1,R2 <RR>

The instruction is suppressed with a specification exception if the contents of arithmetic register R1 are not consistent with a q.ix. It is suppressed with a data exception if pointer register R2 does not identify an ordinary M-space or if the space is in I/O request state, and with an access exception if the space is not in private or family custody of the process within which the instruction is being executed.

A null pointer is loaded into pointer register R2 and into any other pointer register of the process which contains a pointer to the space, and the reference count is decremented by 1 for each pointer loaded. If the reference count does not become zero, the space is placed in system custody, the enqueue request is recorded, and the instruction is completed with condition code 1. The enqueue will be carried out whenever the reference count subsequently becomes zero.

If the reference count is zero, the custody flag is turned off and the M-space is placed at the bottom of the queue, bound to its custody. If the queue is an input queue and was previously empty, an initiation request is triggered designating the process model associated with the queue. If the queue is the null queue, the space is released with the equivalent of a FREE instruction. The instruction is then completed with condition code zero.

Process Class: C,R,D

Condition Code:

0	Space enqueued
1	Enqueuing delayed
2	-
3	-

Exceptions:

Access
Specification
Data

DEQUEUE

DEQ M1,R2 <RR>

If the high order bit of the mask field M1 is zero, the indicated queue is the current queue of the process, otherwise the q.ix is to be found in arithmetic register R2. In that case, the instruction is suppressed with a specification exception if the contents of the arithmetic register do not specify the q.ix of a queue associated with the process.

If the queue is empty or if the indicated queue is the null queue, pointer register R2 is set to the null pointer and the instruction is completed with condition code 1. If the queue is not empty, bit 1 of mask field M1 is examined to determine the domain identifier of the item to be removed from the queue. If the bit is zero, any domain identifier is acceptable; if the bit is 1, the item is to have a domain identifier which matches the identifier of the process within which the instruction is being executed.

Bit 2 of mask field M1 determines the direction of search. If the bit is zero, the item is to be extracted as near to the top of the queue as possible; if the bit is 1, it is to be extracted as near to the bottom as possible. Hence, if bit 1 is zero and bit 2 is also zero, the top item of the queue is removed, while if bit 1 is zero and bit 2 is 1, the bottom item is removed.

If bit 1 is 1, a search of the queue is conducted, starting at the top if bit 2 is zero and at the bottom if bit 2 is 1. The first item found with the proper domain identifier is removed from the queue. If no such item can be found the instruction is terminated with condition code 2.

The low order bit of mask field M1 is examined to determine the protection vector to be assigned to the space removed from the queue. If the bit is zero the space is placed in private custody of the process, with private read and write access; if the bit is 1, the space is placed into family custody, with family read and write access. The custody flag is then turned on, the reference count is set to 1, and a pointer to the space is loaded into pointer register R2.

Bit 2 of field CMFLG of the CMDB for the process model family is examined to determine treatment of the domain identifier of the process. If the bit is zero, the identifier of the process is replaced by that of the space just dequeued. If the bit is 1 the identifier is not replaced. The instruction is then completed with condition code zero.

Process Class: C

Condition Code:

0	Space dequeued
1	Queue empty
2	Space not found
3	-

Exceptions:
Specification

QUEUE SWITCH

QSWCH R2 <RR>

The input queues of the process model for the process are examined to find the queue of highest precedence which is non-empty. That queue becomes the current queue of the process and its q.ix replaces the contents of arithmetic register R2. If there are no input queues for the process or if all queues are empty, the null queue becomes the current queue.

Process Class: C

Condition Code: Unchanged

Exceptions: None

QUEUE WAIT

QWAIT R2 <RR>

The instruction is suppressed with a specification exception if the contents of arithmetic register R2 do not specify the q.ix of a queue associated with the process.

If the queue is non-empty the instruction is immediately completed. If the queue is the null queue, the instruction is interpreted as an IDLE instruction. If the queue is empty, the process is placed into queue wait dispatching condition and the CPU assigned to the process returned to dispatching. The process will be placed into ready condition when an item is entered into the specified queue.

Process Class: C

Condition Code: Unchanged

Exceptions:
Specification

CLOSE GATE

CLOSE D1(B1) <SI>

The gate located by the operand address is made to belong to the process within which the instruction is being executed.

The instruction is suppressed with a specification exception if the gate is not on a word boundary. The instruction is terminated with condition code 3 if the content of the word is not consistent with that of a gate, and with condition code 1 if the gate already belongs to the process.

If the gate is open, it is closed and assigned to the process. The gate count in the space in which the gate is located is incremented by 1, and the instruction is completed with condition code zero. If the gate is closed and belongs to a process of a different process class than the requeusting process, the instruction is terminated with condition code 2.

If the gate is closed and the requesting process is a C-process, the process is placed at the end of the queue of processes waiting for the gate. The PIC is reset to the CLOSE instruction, the process is placed into gate wait dispatching condition, and the CPU assigned to the process is returned to dispatching. The gate wait will be removed by some subsequent OPEN instruction, and the process will again request the gate when next dispatched.

If the gate is closed and the requesting process is an R-process, the process to which the gate belongs is promoted to a dispatching priority fractionally higher than that of the requesting process. The PIC of the requesting process is reset to the CLOSE instruction, the process is suspended, and the CPU assigned to the process is returned to dispatching. The process will again request the gate when next dispatched.

Process Class: C,R
Modal

Condition Code:
0 Gate closed
1 Gate already owned
2 Gate not available
3 Invalid gate format

Exceptions:
Specification

OPEN GATE

OPEN D1(B1) <SI>

The gate located by the operand address is released by the process.

The instruction is suppressed with a specification exception if the gate is not on a word boundary. The instruction is terminated with conditiion code 3 if the content of the word is not consistent with that of a gate, with condition code 2 if the gate is open, and with condition code 1 if the gate does not belong to the process.

The closed gate is opened and the gate count in the space in which it is located is decremented by 1. If the process is a C-process and there is a gate queue, the process at the head of the queue is removed from the queue and taken out of gate wait. The instruction is then completed with condition code zero.

If the process is an R-process which has not been promoted, the instruction is completed with condition code zero. If the process has been promoted it is reduced to its normal priority, suspended, and the CPU assigned to the process is returned to dispatching. The instruction will be completed with condition code zero when the process is next dispatched.

Process Class: C,R
Modal

Condition Code:
0 Gate opened
1 Gate not owned
2 Gate already open
3 Invalid gate format

Exceptions:
Specification

EXIT

EXIT I <RR>

Termination is requested for the process within which the instruction is being executed. Bit 3 of field PFLG of the PIC is set to 1, the byte of immediate data in the I field of the instruction is saved in the state vector, and the instruction is completed by returning the CPU assigned to the process to dispatching.

Termination is completed at some later time by system microcode.

Process Class: C,R

Condition Code: Unchanged

Exceptions: None

TERMINATE PROCESS

TERM D1(B1),I2 <SI>

The process model list is scanned for a process model whose name matches the name stored in the word located by the first operand. If no such model can be found the instruction is terminated with condition code 1. If the process model is not in custody of the family of the process within which the instruction is being executed, or is not in public custody, the instruction is terminated with condition code 2.

Termination is requested for all processes of the family currently in existence. The equivalent of an EXIT instruction with operand field I2 is executed for each process, and the instruction is completed with condition code zero.

Termination is completed at some later time by system microcode.

Process Class: C,R

Condition Code:
0 Request accepted
1 Process model not found
2 Termination not allowed
3 -

Exceptions: None

DEFINE EXCEPTION MODULE

DXM R1,R2 <RR>

The instruction is suppressed with an access exception if pointer register R2 does not identify a module space for which the process has read access.

If the instruction is not suppressed, a pointer to the current exception module of the process is placed into arithmetic register R1, and the instruction is completed by setting the current exception module to be the space identified by pointer register R2.

Process Class: C

Condition Code: Unchanged

Exceptions:
Access

SET EXCEPTION MASK

SXM D1(B1),I2 <SI>

The current exception mask of the process within which the instruction is being executed is stored at the location specified by the first operand. The mask is then set equal to the byte of immediate data in the second operand field.

Process Class: C,R

Condition Code: Unchanged

Exceptions: None

SET BREAKPOINT MASK

SBKM D1(B1),I2 <SI>

The current breakpoint mask of the process within which the instruction is being executed is stored at the location specified by the first operand. The mask is then set equal to the byte of immediate data in the second operand field.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions: None

FORCE PROCESS EXCEPTION

FPX D1(B1),I2 <SI>

The breakpoint mask of the process within which the instruction is being executed is combined with the byte of immediate data in the second operand I2 using a logical AND operation. If the result is zero, the instruction is terminated without further action.

If the result is non-zero, a forced exception record is generated, with the result of the logical and operation stored in the BKP field of the record extension [Figure 5.5]. The pointer portion of the location generated by the first operand is stored in field FXPT of the record extension, and the base address portion of the location is stored in field FXBA. The instruction is then completed by raising a forced exception.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions: Forced by instruction execution

6.0 R-PROCESSES: EVENT RESPONSE

Because R-processes are viewed as primarily event response activities and presumed to have a limited lifetime, they are not provided as much arithmetic capability as C-processes. Moreover, restrictions are imposed for linkage, data access, communication, and in the interpretation of modal instructions that deny to R-processes those functions, such as waiting and dequeuing, which imply unpredictable instruction execution time. The objective of these restrictions is to assure that instruction execution within R-processes will not itself be a source of response delay.

6.1 Process Sources

For any EPSILON system there are three kinds of sources for R-processes:

- system services
  - time-of-day clock
  - system exceptions
- external signals
- I/O devices.

Every source is assigned a dispatching priority [Section 4.6] and a 16-bit identifier as part of the specification of a system, either at the factory or during installation of the source in the field. A source identifier may have any value except zero, which is reserved as a null designation; if the source is an I/O device its source identifier must be the same as its I/O device identifier [Section 7.1]. Apart from these restrictions, the assignment of priority and identifiers is arbitrary. The assignment can be modified at system initialization, but cannot be modified by instruction execution.

The event represented by each source is also set up as part of system specification. Except for the system services, which are fixed sources, any signal from the external signal interface, or any combination of external and I/O device signals which meet the physical configuration constraints of a given EPSILON system model can serve as a source. Processes which are initiated as a result of signals from a source then determine the meaning and significance of the events represented by the source.

The identifiers of all process sources on a given system can be obtained by executing the STORE SOURCE LIST instruction (SSL), which stores source identifiers in successive half-word locations. The number of half-words to be stored is specified in the SSL instruction, and a count of the number of identifiers not stored is returned at instruction completion. The identifiers are stored in order of decreasing dispatching priority, starting with the source of highest priority.

6.2 Process Model Connection

An event can cause process initiation only if a process model is connected to its associated process source. System service sources are connected to built-in process models; however, each model requires some additional information in order to be complete. In some cases the information is supplied using specific instructions (e.g. timing event requests), in other cases it is supplied at system initialization. All other sources are connected by means of the CONNECT R-PROCESS MODEL instruction (CRPM), or the CONNECT R-PROCESS MODEL INDIRECT instruction (CRPMI).

The data for these instructions is supplied by an R-process Model Definition Block (RMDB). An RMDB must begin on a word boundary and have the format described in figure 6.1. The CRPM and CRPMI instructions can be executed only within an R-process; they will connect a new process model, replace an existing one, or delete a model entirely, depending on the content of the RMDB. For the CRPM instruction,

- A new model is connected if there is not already one connected to the specified source. If a model is connected and if deletion is allowed, it will be replaced or deleted; if deletion is not allowed, the connection attempt will be rejected. Replacement occurs if bit 6 of field RMFLG is zero, deletion if the bit is 1. Replacement consists of deleting the existing model, followed by connection of the new model. Deletion will not occur if the connection attempt is rejected for any reason.

- When connection is complete the RMDB area is immediately available for new use, as the process model control information is retained by the system in an area withdrawn from the M-storage pool. The area may be reserved at system initialization; if it is not reserved, connection will be rejected if storage is not available, and the rejection indicated by condition code return.

- The connection attempt will also be rejected if any of the fields RMMOD, RMLOC, RMXMD, or RMCTX are invalid. Field RMMOD must contain either a null pointer or a pointer to a module M-space for which field RMLOC designates an instruction location which lies within the space. Field RMXMD must identify either the null space or a module M-space. The first four bytes of field RMCTX must contain either a null pointer or a pointer to an ordinary M-space in private or family custody of the defining process.

- CRPM is synchronous to the process within which it is executed, so that connection or deletion is effective at instruction completion.

An R-process model is assigned to the custody of the family of the defining process. If the custodian family is deleted from the system the model is transferred to bound custody of the system, or to public custody, according to the value of bit 7 of field RMFLG of its RMDB. It can then be deleted or replaced by execution of a CRPM or CRPMI instruction only if in public custody.

When an R-process model is deleted, spaces held in family custody are deleted with it, as is the entry context space if it is bound to the model. However, if deletion is an intermediate step in replacement of the model, and if the entry context space for the new model is the same as that of the old, the space remains in existence. Its custody is then determined by the value of bit 3 of field RMFLG of the new RMDB. For any deletion sequence, existing processes of the family are not deleted, but will continue activity until terminated by standard termination procedures. The M-storage area used for process model control data will be retained by the system for future use, rather than be returned to the M-storage pool.

CRPMI behaves in the same way as CRPM, except that

- the RMDB data is obtained from another process source rather

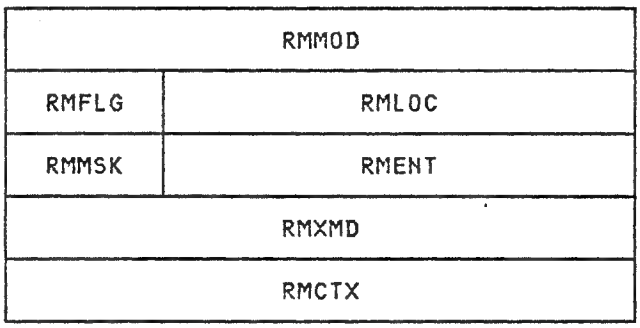

Figure 6.1
R-process Model Definition Block

Field	Offset	Bytes	Description and Use
RMMOD	0	4	A pointer to the module M-space containing the initial instruction sequence to be executed by every process of the process family.
RMFLG	4	1	Bits defining conditions of usage for the process model and members of the family.

Bit	Value	Significance
0	0	Collect process statistics under control of collection instructions
	1	Do not collect process statistics
1	0	The value of the domain identifier for processes of the family is to be the identifier of the entry context space
	1	The value of the domain identifier is to be derived from the process which triggered initiation if it was caused by process action, otherwise it is to be the identifier of the entry context space
2	-	Reserved
3	0	The entry context space identified within field RMCTX is to be bound to the process model
	1	Do not change custody of the entry context space
4	-	Reserved
5	0	This is a single-instance process model
	1	This is a multi-instance process model
6	0	Normal
	1	This process model is to be deleted (is indirectly connected)

7	▮	Place this process model in system custody if custody is transferred
	1	Place in public custody on a transfer of custody

Field	Offset	Bytes	Description and Use
RMLOC	5	3	Base address within the module space identified by field RMMOD of the first instruction of the initial instruction sequence.
RMMSK	▮	1	Value of the exception mask field for initial entry to processes of the family.
RMENT	9	3	Communication data supplied to processes of the family initiated by a source event.
RMXMD	12	4	A pointer to the module M-space containing the initial instruction sequence for handling process exceptions.
RMCTX	16	▮	Entry context data for processes of the family.

---

than from a location in some M-space

- if the instruction results in a model being connected to the source, only the fact of indirect connection is maintained; the RMDB resides with the referenced source, or with the source which was directly connected if there is an indirect connection chain.

The CRPMI instruction is useful when one model is to be connected to several sources, as a change to the model of the directly connected source is effective for all sources indirectly connected to it; moreover, the process requesting the connection is not required to know the content of the RMDB. However, when specific process model conditions are required for a source, a process can execute the STORE SOURCE STATUS instruction (SRCE) to find out if a process model is connected to it. If it is connected, the instruction will supply the RMDB of the connected model; when the RMDB is stored, indirect connection is indicated by turning on the deletion bit of field RMFLG. The stored information can be used to prepare changes to the process modal data.

6.3 Initiation

If a process model is connected to a source, a request for initiation of an R-process is triggered when either

- an event represented by the source occurs, or
- a SIGNAL SOURCE instruction (SGS) is executed specifying the source.

If there are no processes of the family in existence, or if processes exist but the source is not pending, the request causes the source to become pending, and initiation of a member of the family of the connected process model will take place as soon as a CPU is available for it to be dispatched [Section 4.6], provided the M-space specified in field RMMOD

of the RMDB is still in existence. If the M-space has been freed, initiation will be suppressed and an invalid process model system exception will be raised. If the source is pending, treatment of the request depends on its provenance.

- If the signal is from the source, and if there already is a previous source signal waiting to become effective, the request is considered to be satisfied and no further action will be taken. If there is no previous source signal waiting, the request is accepted.

- If the signal is from SGS, and if the operand pointer register identifies the null space, the signal is treated as if it had come from the process source.

- If the operand of an SGS identifies an ordinary space in custody of the requesting process, the request is accepted no matter how many other signals are waiting to become effective. The space is removed from custody of the requesting process, and becomes the **communication space** given to the process initiated as a result of the request.

When a request becomes effective, the initial state vector is loaded with the following standard entry data.

- The PIC contains the location of tbe first instruction to be executed. Field LCUR contains the value of field RMLOC of the RMDB, and field PCUR contains a pointer to the space identified by field RMMOD.

- The condition code is zero if the process was initiated because of a process source event, and 1 if initiation was due to an SGS instruction.

- Pointer register zero contains the entry context space pointer obtained from the first four bytes of field RMCTX; the register is set up as if it had been loaded with an LP instruction. Arithmetic register zero contains entry context data obtained from the last four bytes of field RMCTX.

- Arithmetic register 1 contains the identifier of the process source which triggered initiation. Pointer register 1 contains a null pointer.

- General register 2 contains the communication data. If the condition code is zero, pointer register 2 contains a null pointer, and arithmetic register 2 contains the information supplied in field RMENT of the RMDB or contained in the arithmetic register of the operand of an SGS. If the condition code is 1, general register 2 contains the full communication space pointer and data location submitted with the SGS instruction, divided between arithmetic and pointer registers as if directly transmitted from the general register referenced in the instruction. The pointer register is set up as if it had been loaded by an LP instruction. The space has been transferred to private custody of the process, with its reference count set to 1.

- All other general registers contain zero in the arithmetic register field and a null pointer in the pointer register field.

- The exception mask and domain identifier are set as specified in the process model. If bit 1 of field RMFLG of the RMDB is 1, and if the condition code on en-

try is also 1, the domain identifier of the process is that of the communication space; in all other cases, the identifier is that of the entry context space. The domain identifier of and R-process remains constant throughout the lifetime of the process.

If field PCUR of the PIC identifies a space to which the process does not have read access, the process is immediately terminated and an invalid process model system exception is raised. If the space is the null space, execution is not started. The communication space, if any, is deleted from the system, and termination is requested as if an EXIT instruction with operand value zero had been executed. If the space is not null, instruction execution begins with the instruction located by the PIC and continues until the process completes activity.

6.4 Process Communication

An R-process need not conform to the system view of R-processes as event response activities of relatively short duration, either as a condition of existence or as a means to achieve optimal performance. It may well be better to complete all activity connected with some class of events within the process which is responsible for the initial response to those events. R-processes are therefore provided with many of the facilities of C-processes, and with a substantial amount of basic computational capability.

In general, however, it is advantageous to separate initial response from subsequent computation, particularly when time-constraint criteria are difficult to establish or subject to change. In such a separation, the response period is usually short compared to the computation period, and is confined to activity on data immediately available with the event. Consequently, once an R-process is assigned a CPU and starts running, instruction interpretation continues without time-slicing or explicit waiting periods until terminated by an EXIT or TERM instruction. The CPU may be switched to a process of higher dispatching priority, but if so the switch is a temporary expedient, and a CPU will be re-assigned to the process as soon as possible [Section 4.6]. Moreover, a CPU switch of this kind is invisible to the process; the state vector is preserved intact, and instruction execution will resume without break from the point of suspension.

The absence of time-slicing and explicit waiting capability leads to an instruction execution environment for R-processes which is quite different than that for C-processes. This difference is reflected in the way system functions behave with respect to R-processes, and in the interpretation of modal instructions relating to these functions (e.g. SAVE, LOAD, CALL). The effect is perhaps most noticeable in the area of process communication and synchronization.

- An R-process can send a message to another R-process by placing the data to be transmitted into some space and executing an SGS instruction specifying the source of the receiving process. The location of the message data is specified by the communication data sent with the SGS, or is obtained through some pointer chain anchored by the communication data.

- Because a process source is specified rather than a process, communication is indirect; the semantics imply that the sending process invokes the function

associated with the source, and that function may be carried out by any process of the family. In this respect, communication is similar to the use of queues by C-processes.

- In contrast to C-processes, R-processes cannot expect to exchange a sequence of messages on a continuing basis. SGS always results in the eventual initiation of a new process to receive the data. Moreover, there is no assurance the new process will be initiated unless the sending process terminates; for if it does not, and if there are fewer CPU in the system than R-processes sending messages, initiation cannot occur for the process source of lowest priority involved in the message exchange [Section 4.6].

- Exchange of data between R-processes must therefore be arranged to proceed as in a relay, with a new process taking up the baton at each stage. One method to assure continuity in such an arrangement is for the processes to use a control block interpretation discipline, in which

  - the sending process places its source identifier and re-entry point in a control block associated with the message, and terminates after executing the SGS

  - the receiving process sends a return SGS to the source identified in the control block, and terminates after completing its work

  - the new process of the original source executes a CALL to the re-entry location specified in the control block.

  While this method is generally applicable, specialized relay methods are equally effective, and may well be better for a particular application.

An R-process will use the ENQ instruction to send data to a C-process when computational activity for an event is to begin. In principle, the C-process could use SGS to invoke an R-process for some of the computation, but such usage is dubious. It is likely to fail in practice through lack of data integrity, as an access control gate cannot be used by more than one process class at a time. A C-process should use SGS primarily as a means of simulating the occurrence of some system event.

6.5 Deadlock Detection

All deadlock avoidance techniques depend either on predicting the order in which data will be accessed, or on bounding the domain of access to a point where redundant access control is practical. In an event-driven environment, it is never possible to predict the order in which events will occur, nor is it always possible to place limits on the scope of data access. Consequently, R-process deadlocks are certain to occur in some EPSILON systems, and have a fair probability of occurrence in many systems.

If a group of C-processes become involved in a deadlock, those processes cease to advance but other processes in the system are unaffected. However, if a group of R-processes become deadlocked, and if the number of CPU in the system is less than the number of deadlocked processes, all processes of dispatching priority lower than the highest priority process involved in the deadlock (in particular, all C-processes) will also cease to advance. Dispatching will

not assign a CPU to those processes because the deadlocked processes remain in running condition.

EPSILON systems therefore include a mechanism for detecting R-process deadlocks. The mechanism, which is integrated with R-process dispatching, works in conjunction with the promotion mechanism used to clear an R-process through a closed access control gate [Section 5.10].

- The closure of a sequence of gates by a process can be represented as a graph of directed line segments, where the end points of each segment correspond to gates closed in sequence, and the direction of each segment indicates the order of succession. It is known that if a directed graph of this kind is drawn for a group of processes, then the existence of a circuit in the graph is equivalent to the existence of a deadlock among the processes. Hence, the essential form of a deadlock of two or more processes is that each process tries to close the same set of gates, equal in number to the number of processes, and each gate is first closed by a different process.

- To visualize events that might cause a deadlock, suppose there are processes P1,P2,...,Pn, numbered in order from low to high dispatching priority, and n gates, numbered so that

  P1 will close G1,G2,...,Gn
  P2 will close G2,G3,...,G1
  P3 will close G3,G4,...,G2
  . . . . . .
  . . . . . .
  Pn will close Gn,G1,...,Gn-1.

  P1 is initiated first and closes G1. Before it can close G2, P2 is initated and closes it; before P2 can close G3, P3 is intiated and closes it; before P3 can close G4, P4 is initiated and closes it; this sequence continues until eventually Pn is initiated and closes Gn.

  Since Pn is the highest priority process, it continues to advance until it attempts to close G1. At that point Pn is suspended, process P1 is promoted in priority above it, and if not already running will be assigned a CPU and start running in place of Pn. P1 then causes promotion of P2, which causes promotion of P3, which causes promotion of P4, until eventually Pn is promoted and starts running again. Pn was suspended trying to close G1, so it immediately tries again to close that gate; Pn is suspended as a result of that effort and P1 promoted a second time. P1 then causes promotion of P2, which causes promotion of P3, until eventually Pn is promoted and starts running again. This cycle of promotion continues indefinitely, as all processes are suspended trying to close a gate already closed.

- The occurrence of indefinite promotion, as in the example, characterizes all R-process deadlocks, so it provides the means by which they can be detected. A system parameter, called the **promotion limit**, is defined at system initialization. If an R-process is promoted beyond this limit, it is considered to be party to a deadlock.

The promotion limit specifies the number of times a process can be promoted trying to close any one gate before a deadlock is presumed to be established. If there are N R-process sources in a system with M CPU, then from the R-process dispatching

rules [Section 4.6], at most NM processes can be involved in a deadlock. The number NM+1, called the **ncminal limit**, is therefore an upper bound to the number of promotions which can actually occur before the onset of a deadlock. For any given system, it may be possible to determine a smaller upper bound, so the promotion limit may be set at any value. If no alternative is specified, the system will assume the promotion limit to be equal to the nominal limit.

When a process is promoted beyond the promotion limit, action is initiated to try to recover from the deadlock, if possible.

- Bit 1 of field PFLG in the PIC of the process is set to 1, and an access exception is raised. The exception occurs as soon as the process starts running, so the pending close is not executed.

- Bit 1 is turned off whenever a process opens a gate. Hence, if the exception module is able successfully to back out of a gate, some other process in the deadlocked group will be able to close it. If a deadlock still persists, some process will again exceed the promotion limit and be given an opportunity to open a gate. If all processes which exceed the limit can open a gate, the deadlock will be broken with full recovery.

- If a process exceeds the promotion limit with bit 1 on it is terminated. So, if any of the deadlocked processes have no exception module defined, or the exception module does not open a gate, the gates owned by that process will be opended by termination service [Section 6.6]. In that case, the deadlock will be broken, but the data protected by the gates is likely to have been damaged beyond recovery.

If processes can avoid making irreversible changes in data before closing all access gates required, then recovery from a deadlock is always possible. If irreversible changes cannot be avoided, the effect of the loss of data integrity sustained in a deadlock can only be determined in the context of each application or set of applications.

6.6 Termination

A request for termination of an R-process is triggered by either an EXIT or TERM instruction, both of which behave externally the same as when executed within a C-process [Section 5.12]. The internal microcode which carries out actual deletion of the process does not have the same behavior, as its activity cannot be inserted into a computation cycle.

- A termination request sets bit 3 of field PFLG of the PIC to 1. When this bit is on, dispatching will not assign a CPU to the process if it has been suspended, but will substitute termination service in its place.

- If the termination request is triggered when the process is promoted, bit 3 is set and an OPEN is executed referencing the gate last closed by the process. If termination service becomes active with the process again promoted, an OPEN is executed referring to the gate which caused promotion. In either case, the OPEN will cause the process to be reduced to its normal dispatching priority and suspended.

- Eventually termination service will become active for the process at its normal dispatching priority. If the process has no

I/O requests outstanding at that time, and if it is not holding any access control gates, all spaces in private custody of the process are deleted, the linkage control stack is depleted, the state vector is deleted from M-storage, and the process disappears from the system.

- If it is holding one or more gates, or has I/O requests yet to be completed, the state vector remains in M-storage, but the process is removed from the active list for its source. Each time after that a CLOSE is executed referencing a gate which belongs to the process, an OPEN will be inserted in front of it. Each time an I/O request is completed, the request space is deleted if returned to the process [Section 7.8]. Eventually all gates belonging to the process will be opened and all I/O requests completed; the state vector will then be deleted from M-storage.

Once an R-process is removed from the active list of its source, it does not count to delay a pending condition for the source from becoming effective.

6.7 Exception Handling

The treatment of R-process exceptions with codes 1 through 9 is essentially the same as that for C-processes [Sections 5.13, 5.14]. However, the exception module must be an M-space, and as an R-process cannot execute the DXM instruction, the module must be specified with the process model.

Decimal and floating-point instructions cannot be executed within R-processes, so exception codes 10 through 15 cannot occur. Consequently, as SXM is not a modal instruction, the low-order six bits of the exception mask are available for use by software.

6.8 Instruction Descriptions

STORE SOURCE LIST

SSL R1,D2(X2,B2) <RX>

Source identifiers are stored in successive half-words starting at the second operand location, up to the number of half-words specified by the value contained in arithmetic register R1. Subsequently the content of the register is replaced by the difference between its original value and the number of sources on the system.

Identifiers are stored in order of decreasing dispatching priority, starting with the source of highest priority. If storing an identifier would require exceeding the space boundary, the instruction is terminated at that point with condition code 3, and without decrementing register R1. If the instruction is completed, the condition code is set according to the value of arithmetic register R1.

Process Class: C,R

Condition Code:
0 Difference is zero
1 Difference is negative
2 Difference is positive
3 Insufficient space allowed

Exceptions: None

CONNECT R-PROCESS MODEL

CRPM R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is suppressed with a data exception if arithmetic register R1 does not contain the identifier of a process source. The instruction is terminated with condition code 1 if field RMMOD of the RMDB located by the second operand does not contain either a null pointer or a pointer to a module M-space for which field RMLOC designates a valid instruction location within the space, or if field RMXMD does not contain either a null pointer or a pointer to a module M-space.

If a valid source identifier is contained in R1, and if there is no process model connected to the source, the instruction is terminated with condition code 1 if the first four bytes of field RMCTX do not contain a null pointer or a pointer to an ordinary M-space in private or family custody of the defining process. If the instruction is not terminated, an attempt is made to obtain M-storage for RMDB data. The instruction is terminated with condition code 3 if storage is not available. If storage is available, the RMDB information is stored into it and the instruction completion sequence is entered.

If a process model is already connected to the source, it is tested for deletion status. Deletion is allowed if the model is in custody of the family of the process within which the instruction is being executed, or if the model is in public custody. If deletion is not allowed the instruction is terminated with condition code 2. If deletion is allowed, and if bit 6 of field RMFLG of the proposed new RMDB is 1, the process model data is disconnected from the source, the entry context space is deleted if bound to the model, and the instruction is completed with condition code zero.

If bit 6 of field RMFLG is zero, the RMDB data for the existing model is replaced by the new RMDB data, and the instruction completion sequence is entered. Prior to replacement, the proposed entry context space is compared to the entry context space of the existing model. If the spaces are not the same, and if the space of the existing model is bound to its custody, that space is deleted from the system.

The instruction completion sequence tests bit 3 of field RMFLG to determine the custody of any non-null entry context space. If the bit is zero, the space is bound to the custody of the process model; if the bit is 1, the custody of the space is not altered. The instruction is then completed with condition code zero.

If the process model is deleted or replaced, any active members of its

process model family continue to exist until their termination is specifically requested.

Process Class: R

Condition Code:
0 Model connected or deleted
1 Invalid RMDB format
2 Deletion not allowed
3 Storage not available

Exceptions:
Specification
Data

CONNECT R-PROCESS MODEL INDIRECT

CRPMI R1,R2 <RR>

The instruction is terminated with condition code 3 if either arithmetic register R1 or arithmetic register R2 do not contain process source identifiers.

If both registers contain valid identifiers, the instruction is terminated with condition code 1 if a process model is not connected to the source identified by register R2. If a model is connected with an entry context space which is not bound to it, the instruction is terminated with condition code 1 if the space is not in custody of the process within which the instruction is being executed. If the instruction is not terminated, it is then processed as if it were a CRPM referring to the source identified by the contents of arithmetic register R1, with an RMDB identical to that of the other source.

If the instruction is completed with condition code zero, a reference is generated specifying that the RMDB data of the source identified by register R1 resides with the source identified by register R2. The reference is preserved until execution of another instruction which connects a process model to the source identified by register R1.

Process Class: R

Condition Code:
0 Model connected or deleted
1 Invalid RMDB format
2 Deletion not allowed
3 Source not present

Exceptions: None

STORE SOURCE STATUS

SRCE R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is terminated with condition code 2 if arithmetic register R1 does not contain the identifier of a process source, and with condition code 1 if there is no process model connected to the source.

If the source has a process model connected, the RMDB data for the model is stored into successive words starting at the second operand location. The instruction is then completed with condition code zero.

Process Class: C,R

Condition Code:

0	RMDB stored
1	Process model not connected
2	Source not present
3	-

Exceptions:
Specification

SIGNAL SOURCE

SGS R1,R2 <RR>

The instruction is terminated with condition code 3 if arithmetic register R1 does not contain the identifier of a process source, and with condition code 2 if there is no process model connected to the source.

If a process model is connected, and if pointer register R2 contains a null pointer, the conditions are raised which signal the event associated with the process source. The contents of arithmetic register R2 are retained as communication data to be passed to any process initiated as a result of the signal. The instruction is then completed with condition code zero.

If pointer register R2 identifies a non-null space, the instruction is suppressed with an access exception if the space is not in private or family custody of the process within which the instruction is being executed. If the instruction is not suppressed, a null pointer is loaded into pointer register R2 and into any other pointer register of the process which contains a pointer to the space; the reference count of the space is decremented by 1 for each pointer loaded. The custody flag is turned off, the space is assigned to the source, and the contents of arithmetic register R2 are retained as communication data to be given to the process initiated as a result of the SGS signal.

If the reference count becomes zero, the SGS signal is transmitted to the source and the instruction is completed with condition code zero. If the reference count is not zero, the SGS signal is held in abeyance and the instruction is completed with condition code 1. The SGS signal will be transmitted

whenever the reference count subsequently becomes zero.

Process Class: C,R,D

Condition Code:

0	Signal accepted
1	Signal delayed
2	Process model not connected
3	Source not present

Exceptions:

Access

7.0 D-PROCESSES: INPUT/OUTPUT

This chapter describes I/O operations as they appear to processes, I/O instructions, device control, and D-process functions. The general appearance of the attachment interfaces to I/O devices is discussed, but the publications dedicated to the interfaces should be consulted for a detailed description of interface logical and electrical characteristics.

7.1 I/O Devices

A device is an I/O device of an EPSILON system if it is connected to the system through an I/O attachment interface and conforms to the signalling protocol of that interface. When an I/O device is installed on a system, either at the factory or in the field, it is assigned a unique 16-bit identifier by which it is designated whenever a specific reference to the device is required. A device identifier may have any value except zero, which is reserved as a null device designation. An EPSILON system therefore admits a maximum of 65,535 I/O devices, but some models may restrict configurations to a total substantially less than this.

The actual number and specific identifiers of the I/O devices on a given system can be obtained by execution of a STORE DEVICE LIST instruction (STDL), which stores device identifiers in successive half-word locations. The number of half-words to be stored is specified in the STDL instruction, and a count of the number of identifiers not stored is returned at instruction completion. The identifiers are stored in order of increasing numerical value, starting with the identifier of lowest value.

An I/O device is always a D-process source. It may also be an R-process source if so designed and attached. The conditions under which an I/O device can become an R-process source, and the events it can represent as such a source, are peculiar to the device, and are specified in the individual publications for the device. If an I/O device is both an R-process and D-process source, each source acts independently of the other. The dispatching priority assigned to the device as an R-process source does not affect dispatching of its D-processes, nor is there any special relationship between the processes initiated from the two sources. However, as the I/O device identifier is required to serve for identification of both sources, the sources do have a built-in realtionship which can assist their processes to co-operate.

Any process can obtain information about a device by executing a STORE DEVICE DESCRIPTION instruction (STDDW), which supplies the Device Description Word (DDW) of a specified device. A DDW is a double-word with the format described in figure 7.1. A DDW is generated for each device when it is installed. The device classification combination (CLASS and TYPE) defines the device well enough to distinguish it from all other devices with similar characteristics (e.g. 3330 vs 2314 disc). The DDEP field information is supplied by the device designer to supplement this information; it usually describes specific features peculiar to the device. The format of the DDEP field is therefore dependent on the device classification, and is described in the individual device publications.

Figure 7.2 is a list of the classification combinations so far specified. The list is strictly for

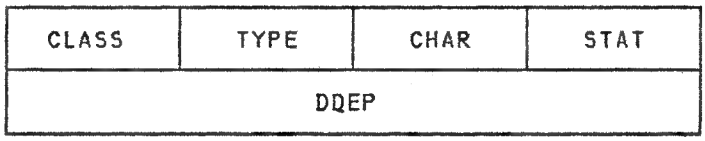

Figure 7.1
Device Description Word

Field	Offset	Bytes	Description and Use
CLASS	0	1	A code specifying the general class of the device.
TYPE	1	1	A code specifying the type of device within the class.
CHAR	2	1	Bits describing characteristics of the device.

Bit	Value	Significance
1	0	This I/O device is not an R-process source
	1	This device is an R-process source
2	0	There is a single I/O path to the device
	1	The device is connected to more than one I/O attachment interface
3-6	-	Reserved
7	0	Normal
	1	Additional device characteristics data available

Field	Offset	Bytes	Description and Use
STAT	3	1	Bits describing the operational status current for the device.

Bit	Value	Significance
0	0	Normal
	1	The device is inaccessible because all paths to it are inoperative
1	0	Normal
	1	The device has been suspended pending repair of damage
2	0	Normal
	1	Intervention required to clear a condition blocking correct device operation

3	0	Normal
	1	I/O operations being held pending a signal from the device
4-5	-	Reserved
6	0	Normal
	1	I/O request pending
7	0	Device not busy
	1	Device busy

Field	Offset	Bytes	Description and Use
DDEP	4	4	Defines specific characteristics within the device class and type.

---

classification, and does not correspond to attachment capability. Devices which appear in the list need not be attachable to any EPSILON system. Devices which can be attached may not appear in the list because classification numbers have not yet been selected.

Bit 7 of field CHAR is turned on to indicate that the device can supply more device dependent descriptive data than will fit into field DDEP. In that case, the device will respond to some SENSE instruction with the additional information. However, as ■ SENSE instruction for a device can only be executed within a D-process of the family associated with the device, device designers should provide the most widely sought information in the DDEP field of the DDW.

7.2 Process Model Connection

Transmission of data between an EPSILON sytstem and an attached I/O device is always controlled by the activity of ■ D-process associated with the device. In order for such a D-process to be initiated, a D-process model must be connected to the device by means of the CONNECT D-PROCESS MODEL instruction (CDPM), or the CONNECT D-PROCESS MODEL INDIRECT instruction (CDPMI). If ■ model is not connected, I/O requests from processes will be rejected, and requests for attention from the device will be ignored.

The connection data for these instructions is supplied by a D-Process Model Definition Block (DMDB). A DMDB must begin on ■ word boundary and have the format described in figure 7.3. The CDPM and CDPMI instructions, which can be executed within any C-process or R-process, will connect a new process model, replace an existing one, or delete a model entirely, depending on the content of the DMDB. For CDPMI the DMDB data is obtained from a model already connected to some device, while for CDPM it is obtained from a location in some M-space.

- A new model is connected if there is not already one connected to the device. If a model is connected and if deletion is allowed, it will be replaced or deleted; if deletion is not allowed, the connection attempt will be rejected. Replacement occurs if bit 6 of field DMFLG is

Class	Type	Device
0		Unspecified
1		Unit record
	1	2540 card reader
	2	2540 card punch
	3	1442/2596 read punch
	8	1403/1404 printer
	16	2671 paper tape reader
	29	1419 mag char reader
	30	1275 optical reader
2		System console
	2	3210 console keyboard
	3	3215 console keyboard
3		Magnetic tape
	1	2400 series tape
	3	3400 series tape
4		DASD
	1	2311 disc storage drive
	2	2301 parallel drum
	5	2321 data cell drive
	8	2314 disc storage
	9	3330 disc storage
5		Communications controller
	2	2701 parallel adapter
	10	3704 controller
	11	3705 controller
	16	3790 subsystem
6		Communications terminal
	1	1050
	3	2740
	5	2741
7		Character display
	3	2260 display station
	9	3270 display system
8		Graphic interactive
	2	2250 display unit

Figure 7.2
Device Classification Combinations

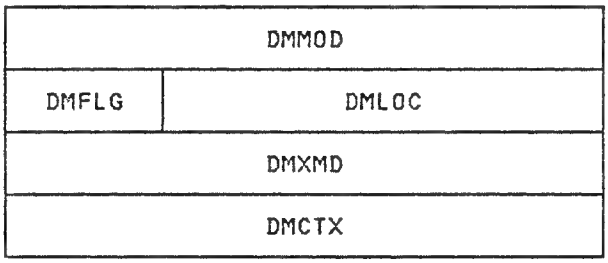

Figure 7.3
D-Process Model Definition Block

Field	Offset	Bytes	Description and Use
DMMOD	0	4	A pointer to the module M-space containing the initial instruction sequence to be executed by every process of the process family.
DMFLG	4	1	Bits defining conditions of usage for the process model and members of the family.

Bit	Value	Significance
0	0	Collect process statistics under control of statistical collection instructions
	1	Do not collect process statistics
1	0	The initial value of the domain identifier for processes of the family is to be the identifier of the entry context space
	1	The initial value of the domain identifier is to be that of the domain for which the device is reserved, or that of the common domain if the device is not reserved
2	0	The domain identifier may vary during execution
	1	The domain identifier is fixed during execution
3	0	The entry context space identified by field DMCTX is to be bound to custody of the process model
	1	Do not change custody of the entry context space
4	0	I/O requests are to be accepted from any process
	1	I/O requests are to be accepted only from processes whose domain identifier matches that of the domain for which the device is reserved
5	0	This device can be reserved for a specified domain
	1	This device cannot be reserved

6	0	Normal
	1	This process model is to be deleted (is indirectly connected)
7	0	Place this process model in system custody if custody is transferred
	1	Place in public custody on a transfer of custody.

Field	Offset	Bytes	Description and Use
DMLOC	5	3	Base address within the module space identified by field DMMOD of the initial instruction sequence to be executed by members of the process model family.
DMXMD	8	4	A pointer to the module space containing the initial instruction sequence for handling process exceptions.
DMCTX	12	4	A pointer to an ordinary M-space which becomes the entry context space for processes of the family.

---

zero, deletion if the bit is 1. Replacement consists of deleting the existing model together with any I/O requests pending on the device, followed by connection of the new model. Deletion will not occur if the connection attempt is rejected for any reason.

- The connection attempt will be rejected if any of the fields DMMOD, DMLOC, DMXMD, or DMCTX are invalid. Field DMMOD must contain either a null pointer or a pointer to a module M-space for which field DMLOC designates an instruction location which lies within the space. Field DMXMD must identify either the null space or a module space. Field DMCTX must contain either a null pointer, or a pointer to an ordinary M-space in private or family custody of the defining process.

- When connection is complete the DMDB area supplied for a CDPM instruction is immediately available for new use, as the process model control information is retained by the system in an area withdrawn from the M-storage pool at system initialization.

- If CDPMI is used and connection or deletion is successful, only the fact of indirect connection is maintained; the DMDB resides with the referenced device, or with the device directly connected if there is an indirect connection chain. Any indirect connection reference is retained until a direct connection is made, irrespective of the connection state of the referenced device.

The STORE DEVICE STATUS instruction (STDS) can be executed within any process to determine if a process model is connected to a device. The instruction will supply the DMDB of the connected model, or an indication that no model is connected. If a DMDB is stored, indirect connection is indicated by turning on the deletion bit of field DMFLG. A connected D-process model is assigned to the custody of the family of the defining process, and can be replaced or deleted only by a custodian pro-

cess. If the custodian family is deleted, the D-process model is transferred to the alternative custody indicated by bit 7 of field DMFLG of its DMDB.

When a D-process model is deleted, spaces held in family custody are deleted with it, as is the entry context space if bound to the model. I/O requests pending on the device are then deleted, followed by deletion of the model. However, if deletion is an intermediate step in replacement of the model, and if the entry context space for the new model is the same as that of the old, the space remains in existence. Its custody is then determined by the value of bit 3 of field DMFLG of the new DMDB.

7.3 I/O Requests

Every I/O device installed on an EPSILON system has a queue associated with it, called the **request queue** for the device. A request queue is considered to be part of the device; it is not distinguished by a separate name or queue index, but is referenced using the device identifier. However, when a D-process model is connected to a device, the request queue becomes associated with the model, and provides the basic functions of an input queue. In particular, an item entered into the queue when the queue is empty automatically triggers a request for initiation of a process of the associated family.

Items are entered into a request queue either by action of the associated device, or by means of the REQUEST INPUT/OUTPUT instruction (RIO). An item is entered for the device when it raises an attention signal through an I/O attachment interface. System microcode responds to the signal by placing a special message in the queue ahead of all ordinary items, but behind any other attention message still resident in the queue. Attention signals therefore take precedence over I/O requests by processes.

RIO is a modal instruction which can be executed within any process. The request will be rejected if the device does not exist or has been suspended for any reason, if there is no D-process model connected to the device, or if the request space is not an ordinary M-space in custody of the requesting process or process family. If RIO is executed within a D-process, the request will also be rejected if the device is the one associated with the process. In addition, domain identification can be used to restrict the set of processes that can make valid requests of a given device [Section 7.9].

If the request is accepted, the M-space specified in the instruction becomes the new bottom item of the queue. As in the case of an ENQ instruction, the new item loses addressability as a space, passing out of the custody of the enqueuing process or process family into the custody of the system. Similarly, the reference count is decremented by substitution of null pointers into pointer registers referencing the space, and entry into the queue is delayed until the count becomes zero. However, in contrast to an input queue, a process can retain a connection to an item in a request queue.

- A space entered into a request queue is placed into a special state, called **request state**, unless the 'no request state' option of the RIO instruction is selected. Request state is assumed to indicate that the activity of the requesting process is in some way dependent on the I/O operation, and that the process intends to remain in existence

until there is some disposition of the request.

- A space placed into request state therefore causes the I/O request count of the process executing the RIO to be incremented. The process is also allowed to reference the space in order to test progress of the request. For that purpose, instead of loading a null pointer into the pointer register actually used to reference the space in the RIO instruction, the retained status of the register is simply changed to indicate that the space is temporarily unavailable. As long as the space remains in request state, the requesting process will receive the 'temporarily unavailable' condition code return from an LP or LPR instruction [Section 3.3].

- A space remains in request state when a D-process serving the request queue acquires custody of it; it is taken out of request state when that process disposes of it with an END I/O REQUEST instruction. Disposition of the request causes the I/O request count of the original requesting process to be decremented, and usually returns the space to custody of that process [Section 7.8].

- Suppression of request state is assumed to indicate that the requesting process is not dependent on the I/O operation, or that communication with the D-process serves some purpose other than I/O. Consequently, the action of the RIO instruction in that case is the same as the action of an ENQ instruction: the I/O request count of the process is not incremented, and all registers referencing the space, including the register designated in the instruction, are loaded with null pointers.

The RIO instruction is synchronous to all processes within which it is executed, but if request state is not suppressed the normal mode is to place C-processes and D-processes into I/O wait state until the request is completed. Processes can always avoid the wait by exercising the 'do not wait' option of the instruction.

A process which is not in I/O wait state can follow the progress of a request by continuing to load the retained pointer to the request space until the condition code indicates the disposition of the space. However, a positive test is provided by the TEST INPUT/OUTPUT instruction (TIO). TIO is a modal instruction which can be executed within any process. When executed within an R-process,

- if the specified space is not in request state, condition code zero indicates that the space has been returned to the custody of the requesting process. Condition code 3 indicates that the space is not in process or family custody; its actual availability can then be tested by an LP or LPR instruction.

- If the space is in request state, the process within which the TIO is being executed must be the requesting process or an access exception will result. If it is the requesting process, condition code 2 is returned if the space is resident in a request queue, and condition code 1 if it is not, indicating the request has been received by some D-process.

  When TIO is executed within a C-process or D-process, the process is placed into I/O wait dis-

patching condition if the request is not complete. The process is removed from I/O wait when the referenced space is disposed of by the D-process servicing the request, at which time the TIO will be retried. Consequently, only condition codes zero or 3 can be returned to a C-process or D-process.

7.4 D-Process Environment

There are no requirements placed on the contents of an M-space entered into a request queue, and no restrictions on the use of the space by the D-process family serving the queue. The presumption is that the space will contain a set of instructions for what is to be done on the device. These instructions are not defined by the architecture, but are arbitrary codes, perhaps privately agreed between the requesting process and the D-process family, perhaps standard to some programming system. A requesting process may also provide data to be used in connection with the request itself; for example, it could include a code indicating how the D-process is to dispose of the request space.

Although this presumptive use of the request space is not necessarily the actual use in any particular case, it does depict the kind of activity a D-process may need to undertake. The activity can extend far beyond simple device control, as the interpretation of an I/O request can include complex data conversion and computation. Consequently, D-processes are provided with a substantial amount of computational capability, though less than that of C-processes or R-processes, and are allowed the ALLOC, FREE, LP, SP, LPIC, CALL, RETURN, ENQ, SGS, RIO, TIO, and breakpoint instructions. However, restrictions are imposed on the D-process environment which limit the scope of activity of any D-process, and constrain the relation between a D-process and a requesting process.

- The state vector of a D-process consists of the 16 general registers, a PIC, and internal control data. The general registers have the same appearance as the general registers of C-processes and R-processes. Register references, register usage, and storage address generation conform to the standard rules [Section 2.8]. However all references to pointer registers 2 through 15 are interpreted as references to pointer register 1. In effect, D-processes have only two independent pointer registers.

- A D-process manages I/O requests specifically for the device with which it is associated, and no other. It is prevented from even trying to manage another device, as the system supplies the device identifier when an identifier is required as an operand of an instruction executable only within D-processes. This method of enforced association also assures that a process model applicable to one device of a class is automatically applicable to all devices of the class.

- Because a D-process and the device with which it is associated are so closely related, D-process models are always single-instance. Any member of the family is expected to dispose of all requests which are in the request queue at the time of initiation, or which are put into

the queue prior to its termination. To reinforce this expectation, all items remaining in the request queue when process termination is triggered are deleted during the termination procedure [Section 7.10].

- In order to allow CPU and PPU to operate independently of one another, transmission of data between an M-space and an I/O device is not subject to access control gate protection, nor is a D-process allowed to execute CLOSE or OPEN instructions. A D-process is expected to be able to carry out its activity without having to share data with other processes, or to take special action to assure the integrity of data in a space not in its private custody. Therefore, if a pointer is stored in a request space for use in data reference by a D-process, or to define a space for data transmission, the requesting process must first obtain the correct access condition for the space.

The D-process family associated with a device represents the device to all other processes. For all practical purposes, processes of the family are the device, as their interpretation of I/O requests determines the appearance and capability the device presents to requesting processes. The restrictions on the D-process environment, and the limitations of the peripheral instruction set, are intended to discourage D-processes from being diverted arbitrarily to general computation or event response.

7.5 Dispatching

D-process dispatching is a fixed function of the system in the sense that neither selection routines nor device priorities can be used to vary the selection algorithm. The algorithm itself is elementary, with primary selection based on attachment of the associated device.

- Every PPU in the system has an internal queue associated with it, called its **dispatching queue**, consisting of D-processes ready to be dispatched, or waiting for a device to complete some operation [Section 7.7]. A PPU will be assigned only to D-processes in its dispatching queue.

- A process will always be placed in the dispatching queue of a PPU whose attachment interface is connected to the device with which the process is associated. If the device is connected to more than one attachment interface, one of the PPU will be selected; the selection is arbitrary, except that a PPU will not be selected if its attachment interface is not operational, or if its path to the device is blocked for some reason.

- A process is placed at the tail of a dispatching queue, unless the entry is in conjunction with a wait condition generated by an uncompleted I/O operation instruction [Section 7.7]. In that case, the process is placed at the head of the queue, or ahead of all processes in the queue which are not in the same waiting condition.

- When a PPU becomes available, its processing mechanism is normally assigned to the process at the head of the queue. However, if the process is in a wait condition, it will be skipped over and the next one in the queue selected.

Thus, apart from conditions which arise because of device delays, pro-

cesses are assigned a PPU in FIFO order as they become ready. As I/O requests are almost all generated by C-processes and R-processes, this algorithm is the simplest one which will meet physical constraints and preserve the priorities generated by process source and computation cycle activity.

7.6 I/O Request Processing

A request for initiation of a D-process is triggered whenever an item is entered into a request queue which was previously empty. If a process of the family associated with the device already exists, the request is considered to be satisfied and no further action will be taken. If a process does not exist, an initial state vector is generated and the new process is entered into the dispatching queue of the appropriate PPU.

When the process is dispatched for the first time, its state vector is set with the following initial data.

- The PIC contains the location of the first instruction to be executed. Field LCUR contains the value of field DMLOC of the DMDB, and field PCUR contains a pointer to the space identified by field DMMOD.

- Pointer register zero contains a pointer to the entry context space; the register is set up as if it had been loaded with an LP instruction. Arithmetic register zero contains zero.

  - Arithmetic register 1 contains the identifier of the device with which the process is associated. Pointer register 1 contains a null pointer.

- All other general registers contain zero in the arithmetic register field and a null pointer in the pointer register field.

- The domain identifier of the process is set to the value specified by the DMDB.

If field PCUR of the PIC identifies a space to which the process does not have read access, the process is terminated immediately and an invalid process model system exception is raised. If the space is the null space, execution is not started. All spaces are removed from the request queue; spaces in request request state are taken out of that state, and spaces not in request state are deleted from the system. Termination of the process is then requested. If the space identified by field PCUR is not null, instruction execution begins with the first instruction and continues until the process terminates or initiates an I/O operation. It is expected that the process will obtain an I/O request from the request queue, interpret the information in the request space, initiate one or more I/O operations as a result of the interpretation, dispose of the request space, obtain another I/O request, and continue this pattern of activity until the request queue is exhausted.

A D-process is not required to conform to this expectation, but as other processes are usually waiting on I/O requests, the D-process is required to conform to the pattern to the extent of providing positive disposition of every I/O request. To enforce this discipline, a request space obtained by a D-process from the request queue is assigned as the **current space** of the process; it remains current until it is taken out of I/O request state. A process cannot obtain a new I/O request as long as it has a space current. If it attempts to terminate with a space still current, that space is returned to the requesting process before ter-

mination occurs [Section 7.10].

The NEXT REQUEST instruction (NEXT) is executed by a D-process to obtain a new I/O request. The instruction is terminated with a condition code indication if the queue is not empty but the process has a request space current. If the queue is empty the instruction indicates either that the process is to be terminated, or that the instruction is to return a condition code for the empty state. If a condition code is generated indicating a new request was not obtained, the PPU assigned to the process will be returned to dispatching for a new assignment. The process will be placed at the tail of its dispatching queue with state vector preserved, so the instruction will actually be completed when the process is next dispatched.

If the request queue is not empty and the process has no current space, the item at the head of the request queue is extracted. If the item was entered by an RIO instruction, a pointer to the request space is placed into the pointer register specified by the instruction. The space is assigned to the private custody of the process, and becomes its current space. If the item was entered into the request queue because of device attention, the space holding the attention data is not assigned as the current space. A D-process need not, therefore, take any overt disposition action for an I/O request originated by its own device.

7.7 Input/Output Operations

There are three instructions which are available to D-processes for the control of I/O operations:

- the START DEVICE instruction (SDV) will initiate an operation on a device
- the HALT DEVICE instruction (HDV) will stop any operation actually in progress
- the WAIT DEVICE instruction (WDV) will cause the process to wait for completion of a specified I/O operation.

The operand of an SDV instruction is a communication block called an I/O Request Block (IORB), which contains the I/O command string which the device is to execute. A command string consists of a device operation, called a **command**, and a set of M-space areas to be used for data transmission. All of the areas must be contained in the same M-space, but their relative locations within the space are not restricted. Each area is defined by a double-word, which contains the base address of the area in the first word, and the extent of the area in a 16-bit field of the second word. The first double-word of a command string contains the command, and each double-word of the string except the last contains a 1 in the first bit of the second word.

In effect, a command string has the format of an S/360 CCW string with data chaining [POP360:105]. The format is illustrated in figure 7.4, in which the ADDR and EXT fields define the base address and extent of the areas. The command contained in field CMND of a command string has the structure of a S/360 command [POP360:105], but with modifier bits (M) and basic code (XX) as described in figure 7.5.

When data is transmitted under control of a command string, the first byte is associated with the base address of the first area, and successive bytes with addresses in ascending order. When the first area is exhausted, the next byte is associated with the base address of the second area, and successive bytes

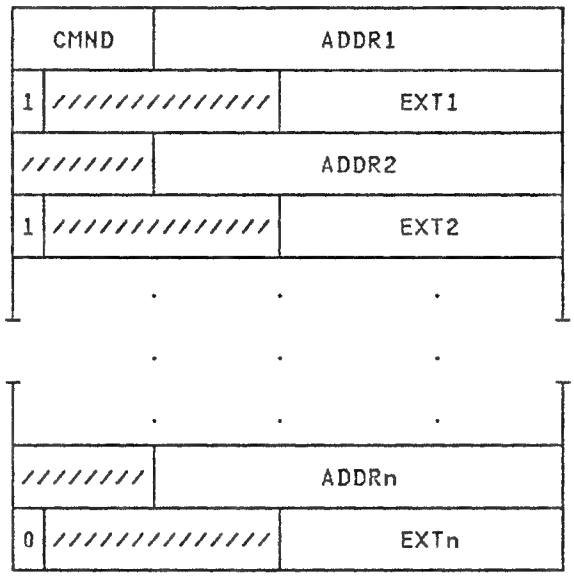

Figure 7.4
Command String Format

Figure 7.5
Command Codes

Command	XX	Description and Use
WRITE	0	Data is to be transmitted from the areas defined by the command string to the device.
READ	1	Data is to be transmitted from the device to the areas defined by the command string.
CONTROL	2	Control information is to be transmitted from the areas defined by the command string to the device. Control information specifies action not involving data transmission. The action may be entirely specified in the modifier bits, or may require additional data. Every device will treat a CONTROL command with all modifier bits zero as a null operation; the device will respond by completing the I/O operation but will take no other action. Apart from the null operation, the format of the control information is peculiar to the device, and is described in device publications.

SENSE 3 Sense information is to be transmitted from the device to the areas defined by the command string. Sense information can describe device characteristics, status, or unusual conditions detected during execution of the previous command string. The amount of sense information supplied varies with the device, as does the format, and details are supplied in device publications. All devices, however, are required to supply the following information in the first byte of sense data transmitted in response to a SENSE command with all modifier bits zero:

Bit	Value	Significance
0	0	Normal
	1	The previous command was rejected because it could not be executed by the device
1	0	Normal
	1	Intervention is required to clear a condition which prevents the device from operating properly (e.g. printer out of paper)
2	0	Normal
	1	The device detected an error in data value during the previous operation which it was unable to correct
3	0	Normal
	1	The device detected an equipment malfunction during the previous operation
4	0	Normal
	1	The device detected an error in data format or recording during the previous operation
5	0	Normal
	1	A timing error occurred during the previous operation which caused incorrect data transmission
6-7	-	Reserved

---

with ascending addresses, and the association continues from area to area until all areas are exhausted. If data transmission is terminated without an exact match between areas and transmitted bytes, the transmission is said to be **inexact.** Inexact transmission occurs if

- an address is generated which lies outside the M-space; transmission stops with the byte which would have been associated with the address
- a device transmits fewer bytes on input than the areas can contain, or the device wants to transmit additional bytes after the areas are filled

- a device expects more bytes on output than are contained in the areas.

Inexact transmission is always noted in the information returned at the completion of an I/O operation, but is not treated as an error.

Part of the information in an IORB is supplied by the process requesting the I/O operation, and the remainder is supplied by the system at operation completion. An IORB must begin on a word boundary and have the format described in figure 7.6.

When an SDV is executed within a D-process, the instruction will be rejected without attemptimg to signal the device if the SPACE field of the IORB does not identify an M-space, or if the D-process does not have the correct I/O access to the space. The I/O access of a D-process to an M-space is defined to be the same as the addressing acces of the D-process to the space, if the process has access to it. If the process does not have access, but has a current request space, then the I/O access of the D-process is the same as the addressing access to the space of the requesting process associated with the current space of the D-process. Using these rules, a READ or SENSE command will be rejected if the process does not have I/O write access, and a WRITE or CONTROL command will be rejected if it does not have I/O read access.

SDV will also be rejected if the device status indicates the operation cannot proceed. Device status is set by execution of a SET DEVICE STATUS instruction (SDS) within a D-process associated with the device, usually when a device error is detected. Four separate conditions are recognized.

- **Inaccessible:** the device is inaccessible because all paths to the device are inoperative.

- **Suspended:** the device has sustained major damage which must be repaired before I/O operations can be resumed.

- **Intervention Required:** operator action is required to clear a condition which prevents the device from operating properly.

- **Not Ready:** the device cannot execute I/O operations until some unspecified temporary condition is cleared.

The setting of these conditions is reflected in the first four bits of field STAT of the DDW [Figure 7.1]. Once set, the conditions can be lifted only by execution of another SDS by a D-process associated with the device. The process usually executes the SDS as a result of an attention signal from the device, or at completion of a SENSE command which indicates a change of status.

If SDV is accepted, an attempt is made to signal the device through the I/O interface. If either the interface or the device is busy, the D-process is placed into **device wait** condition, inserted at the head of its dispatching queue, and the PPU is returned to dispatching. The process is removed from device wait when the busy condition is cleared, and the SDV is retried when the process is next dispatched. This procedure will be repeated, if necessary, until the IORB is accepted by the I/O interface mechanism.

When the IORB is accepted, the process is inserted at the tail of its dispatching queue, and the PPU is returned to dispatching. The process is normally also placed into device wait, and so will not be dispatched again until the operation is com-

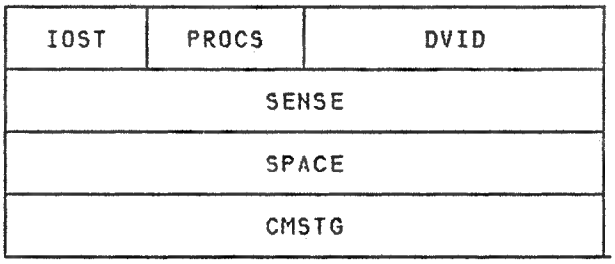

Figure 7.6
I/O Request Block

Field	Offset	Bytes	Description and Use
IOST	0	1	Cleared to zero by the system at the start of the I/O operation. At the end of the operation the system turns on the first bit and sets the other bits to describe the completion status.

Bit	Value	Significance
0	0	The operation is not yet complete
	1	The operation has been completed
1	0	The operation was completed without a device or I/O interface error
	1	A device or I/O interface error was detected
2	0	Sense information has been supplied in field SENSE to describe the error
	1	Sense information has not been supplied for the error
3	0	Normal
	1	Inexact transmission occurred
4	0	Normal
	1	Inexact transmission was due to addressing exception
5-7	-	Reserved

Field	Offset	Bytes	Description and Use
PROCS	1	1	Reserved for arbitrary use by the process. The field is not examined or altered during the course of the I/O operation.
DVID	2	2	The identifier of the device on which the operation occurred. The field is supplied by the system after operation completion unless the SDV instruction specifies otherwise. In that case, the field is available for use by the process, as it will not be

			examined or altered during the course of the operation.
SENSE	4	8	Contains sense information supplied by the system at the completion of any operation for which bit 1 of field IOST is 1 and bit 2 is zero. The information is the same as the first eight bytes of the information which is supplied by the device in response to a SENSE command with all modifier bits zero. If bit 1 of field IOST is zero and bit 3 is 1, the first word of the field contains the address generated when transmission was terminated.
SPACE	12	4	A pointer to the M-space in which the areas defined by the command string reside. The field is supplied by the process.
CMSTG	16	Var	Contains the command string supplied by the process to request the I/O operation.

---

pleted. However, the process can select a 'no wait' option with the SDV instruciton. In that case, it will be dipatched again the next time it arrives at the top of its dispatching queue, irrespective of the state of the I/O operation. A process can select two other options with SDV.

- Sense information is normally supplied with operation completion if there is a device error [Figure 7.6], unless the information could not be obtained for some reason. If the process does not want the standard sense data automatically supplied (it might preclude getting other sense data), the 'no sense data' option can be selected.

- If the process does not require the device identifier to be associated with an operation, it can select the 'no device ID' option. Field DVID of the IORB will then not be set on operation completion, and is free for use by the process.

The WDV instruction will cause a D-process to wait for completion of the I/O operation contained in a specified IORB. If bit zero of field IOST of the IORB is on, the instruction will simply be completed as a null instruction. If the bit is off, the instruction will be rejected if the IORB was not the subject of a previous SDV which initiated an operation not yet complete. If WDV is accepted, the process is placed into device wait, inserted at the tail of its dispatching queue, and the PPU is returned to dispatching. The process will be removed from device wait at the completion of the specified operation.

The HDV instruction will attempt to stop any operation actually in progress on the device, and will delete all operations previously initiated which are not yet complete. If the I/O interface or device adapter are busy, so that the device cannot be signalled, the process is placed into device wait, inserted at the head of its dispatching queue, and the PPU is returned to dispatching. The process is removed from de-

vice wait when the busy condition is cleared, and the HDV is retried when the process is next dispatched. This procedure will be repeated, if necessary, until the device can be signalled to halt the current operation. When the device is halted, or if it was not busy to start with, the instruction is completed by removing from interface control any IORB for the device which was the subject of a previous SDV, and for which the operation has not yet been actually started.

7.8 I/O Request Disposition

When a D-process has completed the interpretation of the information in its current space, it must terminate the I/O request by disposing of that space before another request can be acquired. The normal termination of an I/O request is by means of an END I/O REQUEST instruction (EIOR), which can, in fact, be executed at any time by a D-process. If EIOR is executed when there is no current space, it is treated as a null instruction. If there is a current space,

- the space is removed from I/O request state and the I/O request count of its associated process is decremented by 1. If that process is a C-process or D-process waiting for completion of the request, it is removed from I/O wait dispatching condition.

- The process state vector is altered so that there is no space current for the D-process.

Normal action for EIOR is to return the space to the custody of the requesting process or process family, with the protection vector set to the value in effect at the time the request was made. However, if the 'no return' option of the instruction is chosen, the space will remain in custody of the D-process. In can then be disposed of by a FREE, ENQ, or SGS instruction. Any attempt to reference the space by these instructions while it is still in I/O request state will cause a data exception.

A D-process can also dispose of its current space by means of an RIO instruction. In that case, the request represented by the space has been passed on to another device; the space remains in request state, and its relation to the requesting process is the same as if that process had made the request of the new device rather than the original one.

7.9 Use of Domain Identifier

In contrast to process models of other process classes, a D-process model can be assigned a domain identifier. The domain identifier of a model, which can be different from the domain identifiers assigned to processes of its family, provides a means of reserving devices for specific use, if desired. A device is said to be **reserved** if bit 4 of field DMFLG of the DMDB of the process model connected to the device is set to 1. If a device is reserved for any domain except the common domain, I/O requests are accepted only from those processes whose domain identifier matches the domain identifier of the D-process model. A device reserved for the common domain is treated as if it were not reserved.

The initial reservation of a device becomes effective as soon as a process model is connected or replaced.

- If bit 4 of field DMFLG or the DMDB is zero, the device is not reserved.

- If bit 4 is 1, the device is reserved for the domain to which the entry context space belongs.

If bit 3 of field DMFLG of the DMDB so indicates, the identifier can be replaced by execution of a RESERVE DEVICE instruction (RSRV). RSRV can be executed within any process which can replace or delete the process model; it becomes effective as soon as it is executed, whether or not a D-process of the family exists.

The rules for acquisition of domain identifier by a D-process are similar to those for a C-process [Section 5.9].

- If bit 1 of field DMFLG of the DMDB is zero, the initial domain identifier of a D-process is the domain identifier of its entry context space.

- If the bit is 1, the initial domain identifer is that of the domain for which the associated device is reserved, or that of the common domain if the device is not reserved.

- If bit 2 of field DMFLG is 1, the initial identifier is retained by the process throughout its lifetime. If bit 2 is zero, the domain identifier is changed by execution of a NEXT instruction to the domain identifier of the space obtained from the request queue.

If a device reservation is altered before all I/O requests for a previous reservation have been completed, resource usage statistics may be logged under a different identifier than the one which was used to admit the requests. This situation is not considered to be an error, as reservation and resource utiliztion are accounted for separately.

7.10 Termination

A request for termination of a D-process is triggered by an END PROCESS instruction (END). The request can signify normal completion, or recognition of a condition for the process or device which precludes continuation. END is similar to EXIT in that it sets up an entry to a service which will complete essential procedures left uncompleted by the process, and will recover resources before actually deleting the process from the system. Termination service gets control first when the process is next dispatched after completion of the END.

- if the process has a current space, termination service first disposes of the space as if an EIOR which returns the space to its original custody had been inserted in the instruction stream prior to the END.

- Requests which remain in the request queue are deleted as if they, too, had been subjected to EIOR. When all requests have been deleted, the linkage control stack of the process is removed by completing any open return sequences.

- If the process has no I/O requests outstanding, all spaces which remain in its private custody are deleted, and the process is deleted from the system. If there are outstanding requests, the process is inserted at the tail of its dispatching queue, and the PPU assigned to the process is returned to dispatching.

If process termination is not completed with the first entry, termination service will continue to insert the process at the tail of its dispatching queue until it gets control with all I/O requests completed; it will then complete termination of the process. If an initiation request is triggered prior to completion of termination, the request

is left in the request queue, and a new process is initiated at once to replace the terminating process.

7.11 Exception Handling

D-process have a fixed exception mask which is set to zero, so that all class 4 exceptions are suppressed. The treatment of D-process exceptions with codes 1 through 7 is essentially the same as that for C-processes [Sections 5.13, 5.14], except that as a D-process cannot execute the DXM instruction, any exception module must be specified with the process model.

If an invalid process model system exception is raised because field DMXMD of the DMDB identifies a space no longer in existence, the process is terminated even for a class 3 exception.

7.12 Attachment Interfaces

A device actually connected to an I/O interface is usually an adapter which converts the interface signals to the signals and formats expected by its devices. Adapters can be integral with a single device or can themselves interface to a number of devices. The purpose of an adapter may be device control only, or it may alter the appearance of the interface to that expected by devices designed for other interfaces (e.g. the S/370 channels). The only requirements on adapters are that they conform to the interface signals and protocol.

A channel I/O interface contains polling, selection, and control lines, as well as a data bus. The interface electrical lines are passed from device to device to establish a multi-drop connection. Devices monitor the state of these lines by attaching them to receivers, and raise signals on the lines by connecting line drivers. An EPSILON channel is therefore similar to an S/370 channel, in that devices are interlocked by DC signals which are passed from one device to the next for polling, selection, and control.

A loop I/O interface, however, is not interlocked by DC signals. Devices connected to a loop receive a continuous stream of signals representing bits organized into groups called **frames**. Frames contain both control information and data, the content and destination being determined by the frame itself of by the content of previous frames. Frame discipline for EPSILON loops is designed to minimize the number of bits required to convey information on a loop.

Every device on an EPSILON system is assigned a selection address which designates its attachment interface, and uniquely distinguishes it from all other devices connected to the same interface. Selection addresses are assigned as devices are connected to a system, and can have any 16-bit value. The relation between device identifier and selection address is available only to system microcode, which uses this information both to signal the correct device and to conform to the proper interface protocol. D-processes are therefore not required to take into account the kind of interface to which a device is attached.

7.13 Instruction Descriptions

STORE DEVICE LIST

STDL R1,D2(X2,B2) <RX>

Device identifiers are stored in successive half-words starting at the second operand location, up to the number of half-words specified by the value contained in arithmetic register R1. Subsequently the content of the register is replaced by the difference between its original value and the number of devices connected to the system.

Identifiers are stored in order of increasing numerical value, starting with the identifier of lowest value. If storing an identifier would require exceeding the space boundary, the instruction is terminated with condition code 3, and without decrementing register R1. If the instruction is completed, the condition code is set according to the value of arithmetic register R1.

Process Class: C,R

Condition Code:

0	Difference is zero
1	Difference is negative
2	Difference is positive
3	Insufficient space allowed

Exceptions: None

STORE DEVICE DESCRIPTION

STDDW R1,D2(B2) <RS>

The DDW of the device whose identifer is contained in arithmetic register R1 is stored at the second operand location.

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is terminated with conditon code 1 if register R1 does not contain the identifier of a device connected to the system.

Process Class: C,R,D

Condition Code:

0	DDW stored
1	Device not present
2	-
3	-

Exceptions:
Specification

CONNECT D-PROCESS MODEL

CDPM R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is suppressed with a data exception if arithmetic register R1 does not contain the identifier of a device connected to the system. The instruction is terminated with condition code 1 if field DMMOD of the DMDB located by the second operand does not contain either a null pointer or a pointer to a module M-space for which field DMLOC designates a valid instruction location within the space, or if field DMXMD does not contain either a null pointer of a pointer to a module space.

If there is no process model connected to the device, the instruction is terminated with condition code 1 if field DMCTX does not contain a null pointer of a pointer to an ordinary M-space in private or family custody of the process within which the instruction is being executed. If the instruction is not terminated, the proposed model is connected to the device, and assigned to the custody of the family of the process within which the instruction is being executed. The instruction completion sequence is then entered.

If a process model is already connected to the device, it is tested for deletion status. Deletion is allowed if the model is in custody of the family of the process within which the instruction is being executed or if the model is in public custody. The instruction is terminated with condition code 2 if deletion is not allowed. If deletion is allowed, and if field DMMOD of the proposed new DMDB contains a null pointer, the process model data is disconnected from the device, the entry context space is deleted if bound to the model, and the instruction is completed with condition code zero.

If field DMMOD is non-null, the DMDB data for the existing model is replaced by the new DMDB data, and the instruction completion sequence is entered. Prior to replacement, the proposed entry context space is compared to the entry context space of the existing model. If the spaces are not the same, and if the space of the existing model is bound to its custody, that space is deleted from the system.

The instruction completion sequence tests bit 3 of field DMFLG to determine the custody of any non-null entry context space. If the bit is zero, the space is bound to custody of the process model; if the bit is 1, the custody of the space is not altered. The instruction is then completed with conditon code zero.

If the process model is deleted or replaced when there is a process of the family active, that process continues to exist until its termination is specifically requested.

Process Class: C,R

Condition Code:
0 Model connected or deleted
1 Invalid DMDB format
2 Deletion not allowed
3 -

Exceptions:
Specification
Data

CONNECT D-PROCESS MODEL INDIRECT

CDPMI R1,R2 <RR>

The instruction is terminated with condition code 3 if either arithmetic register R1 or arithmetic register R2 do not contain the identifier of a device connected to the system.

If both registers contain valid identifiers, the instruction is terminated with condition code 1 if a process model is not is not connected to the device identified by register R2. If a model is connected with an entry context space which is not bound to it, the instruction is terminated with condition code 1 if the space is not in private or family custody of the process within which the instruction is being executed. If the instruction is not terminated, it is then processed as if it were a CDPM referring to the device identified by the contents of register R1, with a DMDB identical to that of the other device.

If the instruction is completed with condition code zero, a reference is generated specifying that the DMDB data of the device identified by register R1 resides with the device identified by register R2. The reference is preserved until execution of another instruction which connects a process model to the the device identified by register R1.

Process Class: C,R

Condition Code:
0 Model connected or deleted
1 Invalid DMDB format
2 Deletion not allowed
3 Device not present

Exceptions: None

STORE DEVICE STATUS

STDS R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is terminated with condition code 2 if arithmetic register R1 does not contain a device identifier, and with condition code 1 if a process model is not connected to the device.

If the device has a process model connected, the DMDB data for the model is stored into successive words, starting at the second operand location. The instruction is then completed with condition code zero.

Process Class: C,R

Condition Code:

0	DMDB stored
1	Process model not connected
2	Device not present
3	-

Exceptions:
Specification

REQUEST INPUT/OUTPUT

RIO M1,R2 <RR>

The instruction is suppressed with a data exception if pointer register R2 does not identify an ordinary M-space, and with an access exception if the space is not in private or family custody of the process within which the instruction is being executed. It is terminated with condition code 3 if arithmetic register R2 does not contain the identifier of a device connected to the system, with condition code 2 if the device is inaccessible or suspended, and with condition code 1 if there is no process model connected to the device. If the instruction is being executed within a D-process, it will also be terminated with condition code 3 if register R1 contains the identifier of the device associated with the process.

If the instruction is not suppressed or terminated, the reference count of the space is decremented by 1. The high order bit of mask field M1 is examined to determine if the space is to be placed into request state. If the bit is zero, the retained status of pointer register R2 is set to 'space not available', and the space is placed into request state by recording the protection vector, the requesting process, and the device on which the request was made. If the space was already in request state, the device is recorded without altering the other recorded data. If the bit is 1, a null pointer is loaded into pointer register R2, and request state data is not recorded.

A null pointer is then loaded into any other pointer register of the process which contains a pointer to the space, and the reference count is decre-

mented by 1 for each pointer loaded. If the reference count coes not become zero, the space is placed in system custody, the request is recorded, and the instruction completion sequence is entered. The space will be placed into the request queue for the device whenever the count subsequently becomes zero.

If the reference count is zero, the custody flag is turned off, and the space is inserted at the bottom of the request queue for the device, bound to its custody. If the request queue was empty at the time of insertion, an initiation request is triggered designating the process model connected to the device. The completion sequence is then carried out.

If the instruction is being executed within an R-process, it is completed by setting the condition code to zero. If the process is a C-process or D-process, if the space was placed onto request state, and if bit 1 of mask field M1 is zero, the process is placed into I/O wait dispatching condition. The I/O wait condition is not generated if bit 1 is 1, or if request state is suppressed. The instruction is then completed with condition code zero.

If an I/O wait condition is generated, the processing mechanism assigned to the process is returned to dispatching. The wait condition will be removed when the space is removed from request state. In the case of a D-process, completion also includes inserting the process at the tail of its dispatching queue.

Process Class: C,R,D
Modal

Condition Code:
0 Request accepted
1 Process model not connected
2 Device not available
3 Device not present

Exceptions:
Access
Data

TEST INPUT/OUTPUT

TIO R2 <RR>

The space identified by pointer register R2 is examined for request state. If the space is not in request state and not in custody of the process or process family within which the instruction is being executed, the instruction is completed with condition code 3, otherwise it is completed with condition code zero.

If the space is in request state, and if it is the current space of the process associated with the device on which the request was made, condition code 1 is set and the completion sequence is entered. If the space is not the current space, condition code 2 is set.

If the instruction is being executed within an R-process, it is completed as soon as the condition code is set. If the process is a C-process or D-process, the process is placed into I/O wait dispatching condition, and the processing mechanism assigned to the process is returned to dispatching. The

wait will be removed when the space identified by pointer register R2 is removed from request state. In the case of a D-process, completion also includes inserting the process at the tail of its dispatching queue.

Process Class: C,R,D
Modal

Condition Code:
0 Request complete
1 Request in progress
2 Request enqueued
3 Request not found

Exceptions: None

NEXT REQUEST

NEXT M1,R2 <RR>

If the process within which the instruction is being executed has a current space, the instruction is terminated with condition code 2. If the request queue for the device associated with the process is empty, and if the high-order bit of mask field M1 is zero, process termination is requested. If the bit is not zero, the instruction is terminated with condition code 1.

If the request queue is not empty, the item at the head of the queue is removed, a pointer to it is loaded into pointer register R2, its custody flag is turned on, and its reference count is set to 1. The space is placed into the private custody of the process, and if it is in request state, it becomes the current space of the process.

Bit 2 of field DMFLG of the DMDB for the process model of the family is examined to determine treatment of the domain identifier of the process. If the bit is zero, the domain identifier of the process is replaced by the domain identifier of the space just removed from the request queue. If the bit is 1, the domain identifier of the process is not altered. The instruction is then completed with condition code zero.

If the instruction is terminated, the process is inserted at the tail of its dispatching queue, and the PPU is returned to dispatching. The condition code return will become effective when the process is next dispatched.

Process Class: D

Condition Code:
0 Request obtained
1 Request queue empty
2 Space still current
3 -

Exceptions: None

START DEVICE

SDV M1,D2(X2,B2) <RX>

The instruction is suppressed with a specification exception if the second operand does not locate a word boundary. It is terminated with condition code 3 if the device associated with the process is inacessible, suspended, not ready, or requires intervention.

If the space identified by field SPACE of the IORB located by the second operand is not an ordinary M-space, the instruction is terminated with condition code 2. If the space is not in custody of the process and is not in request state, the instruction is terminated with condition code 1. If the space is not in custody of the process but is in request state, the access to the space of the requesting process is tested for validity. If the command in field CMSTG of the IORB is a READ or SENSE, the process must have write access to be valid. If the command is WRITE or CONTROL the process must have read access to be valid. The instruction is terminated with condition code 1 if the access is not valid.

If the access is valid, a signal is sent to the I/O interface requesting acceptance of the IORB for the device with which the process is associated. If the interface or device is busy, the process is placed into device wait condition, inserted at the head of its dispatching queue, and the PPU returned to dispatching. The wait will be removed when the busy condition which caused it is removed, and the instruction will be retried when the process next gets dispatched.

If the IORB is accepted, the process is inserted at the tail of its dispatching queue. If the high-order bit of mask field M1 is zero, the process is also placed into device wait dispatching condition. Condition code zero is then set and the instruction is completed by returning the PPU to dispatching.

The mask field is transmitted to the I/O interface with the location of the IORB, for use by the operation completion sequence. If bit 1 of the field is zero and the operation is completed with device error, sense data will be obtained from the device and stored into field SENSE of the IORB. Up to eight bytes of data will be stored, depending on the response of the device to a SENSE command with modifier bits all zero. If bit 1 is not zero, sense data will not be obtained. If bit 2 is zero, the identifier of the device will be stored into field DVID of the IORB, otherwise it will not. The remaining bits of the mask field are not examined.

Process Class: D

Condition Code:

0	Operation accepted
1	Invalid access
2	Invalid space
3	Device not available

Exceptions:
Specification

WAIT DEVICE

WDV D2(X2,B2) <RX>

The instruction is suppressed with a specification exception if the second operand does not locate a word boundary. It is terminated with condition code 3 if the device associated with the process is inaccessible, suspended, not ready, or requires intervention.

The IORB located by the second operand is tested for acceptance by the I/O interface. If acceptance is not recorded and bit zero of field IOST is set to 1, the instruction is completed with condition code zero. If the bit it zero, the instruciton is terminated with condition code 1.

If the IORB acceptance is still recorded, the process is placed into device wait dispatching condition and inserted at the tail of its dispatching queue. Condition code zero is set, and the PPU is returned to dispatching. The process will be removed from device wait when the operation contained in the IORB located by the second operand is completed.

Process Class: D

Condition Code:

0	Operation complete
1	Operation unknown
2	-
3	Device not available

Exceptions:
Specification

HALT DEVICE

HDV R2 <RR>

The instruction is terminated with condition code 3 if the device associated with the process is inaccessible, suspended, not ready, or requires intervention.

If the device is available, a signal is sent to the I/O interface requesting the device be halted. If the interface or device adapter are busy, the process is placed in device wait dispatching conditon, inserted at the head of its dispatching queue, and the PPU returned to dispatching. The wait will be removed when the busy condition which caused it is removed, and the instruction will be retried when the process is next dispatched.

If the signal is accepted, the device is signalled to halt. If the device was busy with an operation contained in an IORB, the location of the IORB is placed in general register R2 and condiion code zero is set. If it was not engaged in an operation contained in an IORB, general register R2 is not altered, and condition code 1 is set.

The instruction is then completed by requesting cancellation by the I/O interface of any IORB for the device still recorded as accepted. Bits zero

and 3 of field IOST are set to 1 for cancelled IORB.

Process Class: D

Condition Code:
0 IORB location loaded
1 IORB location not loaded
2 -
3 Device not available

Exceptions: None

SET DEVICE STATUS

SDS M1,M2 <RR>

The status of the device associated with the process is set according to the bits of the mask fields.

Mask field M2 indicates which conditions are to be altered. The bits correspond in high-to-low order to the conditions inaccessible, suspended, intervention required, and not ready. If a bit is zero the corresponding condition is to remain as is; if a bit is 1 the condition is to be set to the value indicated by mask field M1.

The bits in mask field M1 correspond to the conditions in the same order. A bit specifies the condition to be on (set) if its value is 1, and to be off (reset) if its value is zero.

Process Class: D

Condition Code: Unchanged

Exceptions: None

END I/O REQUEST

EIOR M1 <RR>

If the process has no space current, the instruction is terminated with condition code 2. If there is a current space, it is removed from request state and the I/O request count of the associated process is decremented by 1. If that process is a C-process or D-process in I/O wait dispatching condition, it is removed from the wait condition.

If bit zero of mask field M1 is zero, the space is returned to the custody of the requesting process or process family, with the protection vector set to the value preserved when the space was placed into request state. The pointer registers of the process within which the instruction is being executed are examined. If a register contains a pointer to the space, it is loaded with the null pointer. The reference count of the space is decremented by 1 for every null pointer loaded. Condition code zero is set for completion.

If bit zero of mask field M1 is not zero, the space is left in custody of the process executing the instruction, and condition code 1 is set for completion.

The process is then placed in the condtion of having no space current, and the instruction is completed with the condtion code previously set.

Process Class: D

Condition Code:

0 Space returned
1 Space retained
2 No space was current
3 -

Exceptions: None

RESERVE DEVICE

RSRV R1,R2 <RR>

The instruction is terminated with condition code 3 if arithmetic register R1 does not identify a device connected to the system, or if a process model is not connected to the device. It is suppressed with a data exception if arithmetic register R2 does not contain a valid device identifier.

If a process model is connected, the instruction is terminated with condition code 2 if the model is not in public custody or in custody of the process or process family within which the instruction is being executed. It is terminated with condition code 1 if custody is acceptable but bit 5 of field DMFLG of the DMDB indicates that the device cannot be reserved.

If the device can be reserved, the process model is assigned the domain identifier contained in arithmetic register R2. The instruction is then completed with condition code zero.

Process Class: C,R

Condition Code:

0 Device reserved
1 Reservation not allowed
2 Invalid process model reference
3 Process model not connected

Exceptions:

Data

END PROCESS

END I <RR>

Termination is requested for the process within which the instruction is being executed. The byte of immediate data is stored in the state vector, the process is inserted at the tail of its dispatching queue, and the instruction is completed by returning the PPU assigned to the process to dispatching.

Termination is completed at some later time by system microcode [Section 7.10].

Process Class: D

Condition Code: Unchanged

Exceptions: None

8.0 GENERAL INSTRUCTIONS

Most of the EPSILON system arithmetic, logical, and branching instructions have been adopted from the S/360 and S/370 non-privileged instruction sets, with only minor changes of interpretation. Therefore, the following conventions have been employed as a means of minimizing the amount of material necessary to describe these instructions.

- The instructions are organized into groups related by class of function or by some characteristic of interpretation.

- Each group is introduced by a general discussion of the operand field interpretation for the instruction formats which apply to the group, and a description of operand data formats. This discussion is followed by a table listing all instructions of the group.

- If the column in the table headed 'description' contains an entry which refers to POP360 or POP370, the corresponding instruction behaves exactly as described by the reference, apart from any changes of interpretation which apply to the whole group. In that case, no further description of the instruction appears in this document. If the column entry is the word 'new', the instruction is new, while if the entry is blank the instruction differs from the S/360 or S/370 instruction in some non-superficial way. In either case, a full description of the instruction follows the table.

The table for each group also lists the instruction mnemonic and process class. The operation codes, which are not listed, are the same as the codes for the corresponding S/360 or S/370 instruction, if there is one. Operation codes for new instructions have not yet been selected.

8.1 Fixed-Point Arithmetic

Fixed-point arithmetic is essentially the same as S/360 and S/370. Numbers are represented as integers in two's-complement form, either 16, 32, or 64 bits in extent. The first bit of a number field is considered to be a sign bit, except that for instructions with the word LOGICAL in their name the entire field is treated as an unsigned integer.

Fixed-point instructions use the RR, RX, and RS formats. In these formats, whose operand fields are represented in symbolic assembler notation as

```
R1,R2
R1,D2(X2,B2)
R1,R3,D2(B2)
```

the first and third operands always designate an arithmetic register. The second operand may specify a register or a location, depending on the format and instruction.

In the RR format, the second operand field, R2, designates an arithmetic register which contains the actual operand data. In the RX format, the B2 field designates a general register whose contents, together with the displacement D2 and the contents of the arithmetic register designated by X2, are combined using the rules of Section 2.8 to form the M-space address of the operand data. In the RS format, the B2 field of the instructions LOAD MULTIPLE and STORE MULTIPLE designates a general register which is used to form the second operand address as in the RX format,

except that indexing cannot occur. In the shift instructions, the B2 field designates an arithmetic register whose contents are added to the D2 field to form a sum which specifies the amount of the shift.

Instructions can designate the same register in all operand fields with results consistent with using different registers, as address computation is completed before instruction execution. Instruction results replace the first (and third) operands, except for the store instructions for which the result replaces the second operand.

Figure 8.1
Fixed-Point Arithmetic Instructions

Name	Symbol	Type	Class	Description
ADD	AR	RR	C,R,D	POP370:122
ADD	A	RX	C,R	POP370:122
ADD HALFWORD	AH	RX	C,R	POP370:122
ADD LOGICAL	ALR	RR	C,R,D	POP370:122
ADD LOGICAL	AL	RX	C,R	POP370:122
COMPARE	CR	RR	C,R,D	POP370:126
COMPARE	C	RX	C,R	POP370:126
COMPARE HALFWORD	CH	RX	C,R	POP370:128
COMPARE LOGICAL	CLR	RR	C,R,D	POP370:128
COMPARE LOGICAL	CL	RX	C,R	POP370:128
DIVIDE	DR	RR	C,R,D	POP370:131
DIVIDE	D	RX	C,R	POP370:131
LOAD	LR	RR	C,R,D	POP370:134
LOAD	L	RX	C,R,D	POP370:134
LOAD AND TEST	LTR	RR	C,R,D	POP370:134
LOAD AND TEST	LT	RX	C,R,D	New
LOAD COMPLEMENT	LCR	RR	C,R,D	POP370:134
LOAD HALFWORD	LH	RX	C,R	POP370:135
LOAD MULTIPLE	LM	RS	C,R,D	POP370:135
LOAD NEGATIVE	LNR	RR	C,R,D	POP370:135
LOAD POSITIVE	LPR	RR	C,R,D	POP370:135
MULTIPLY	MR	RR	C,R,D	POP370:139
MULTIPLY	M	RX	C,R	POP370:139
MULTIPLY HALFWORD	MH	RX	C,R	POP370:140
SHIFT LEFT DOUBLE	SLDA	RS	C,R,D	POP370:141
SHIFT LEFT DOUBLE LOGICAL	SLDL	RS	C,R,D	POP370:142
SHIFT LEFT SINGLE	SLA	RS	C,R,D	POP370:142
SHIFT LEFT SINGLE LOGICAL	SLL	RS	C,R,D	POP370:143
SHIFT RIGHT DOUBLE	SRDA	RS	C,R,D	POP370:143
SHIFT RIGHT DOUBLE LOGICAL	SRDL	RS	C,R,D	POP370:143
SHIFT RIGHT SINGLE	SRA	RS	C,R,D	POP370:144
SHIFT RIGHT SINGLE LOGICAL	SRL	RS	C,R,D	POP370:144
STORE	ST	RX	C,R,D	POP370:144
STORE HALFWORD	STH	RX	C,R	POP370:146
STORE MULTIPLE	STM	RS	C,R,D	POP370:146
SUBTRACT	SR	RR	C,R,D	POP370:146

SUBTRACT	S	RX	C,R	POP370:146
SUBTRACT HALFWORD	SH	RX	C,R	POP370:147
SUBTRACT LOGICAL	SLR	RR	C,R,D	POP370:147
SUBTRACT LOGICAL	SL	RX	C,R	POP370:147

LOAD AND TEST

LT R1,D2(X2,B2) <RX>

The second operand is placed unchanged into arithmetic register R1, and the sign and magnitude of the result determine the condition code.

Process Class: C,R,D

Condition Code:
0 Result is zero
1 Result is less than zero
2 Result is greater than zero
3 -

Exceptions:
Access

8.2 Logical Operations

Logical operations manipulate data as uniform bit fields of fixed length, or as a variable length string of character bytes. Fixed length data fields consist of a single or double word, or a single character. As instruction operands, they are resident in arithmetic registers, or an M-space, or are extracted from a field in the instruction. Variable length data fields always reside in an M-space, and are acted upon by instructions which relate them to some other variable length field, not necessarily in the same space.

The logical operation instructions use all five instruction formats, RR, RX, RS, SI, and SS. In the RR, RX, and RS formats, the first operand field always designates an arithmetic register, as does the second operand field of the RR format. In the RX and RS formats, the B2 field designates a general register whose contnents, together with the displacement D2 and the contents of the arithmetic register designated by X2 (if present), are combined using the rules of Section 2.8 to form the M-space address of the second operand data. In the RS format, the R3 field is used as a mask to identify those bytes of the first operand data actually acted upon by the instruction.

For the SI and SS formats, the operand fields are represented in symbolic assembler notation as

D1(B1),I2
D1(L,B1),D2(B2)

The B1 field designates a general register whose contents are combined with the displacement D1 to form the M-space location of the first operand data. In the SS format, the L field is a byte whose value specifies the

length of the operand data fields, and the B2 field designates a general register whose contents are combined with the displacement D2 to form the M-space location of the second operand data. In the SI format, the I2 field is a byte which itself is the second operand data.

Instructions can designate the same register in all operand fields with results consistent with using different registers, as address computation is completed before instruction execution. In the SS format instructions, the operand data fields can overlap, with results consistent with handling one byte at a time, starting at the address of the first byte of the first operand. Results replace the the first operand, except for store instructions for which the result replaces the second operand.

Name	Symbol	Type	Class	Description
AND	NR	RR	C,R,D	POP370:123
AND	N	RX	C,R	POP370:123
AND CHARACTER	NC	SS	C	POP370:123
AND IMMEDIATE	NI	SI	C,R,D	POP370:123
COMPARE LOGICAL CHARACTER	CLC	SS	C	POP370:128
COMPARE LOGICAL IMMEDIATE	CLI	SI	C,R,D	POP370:128
COMPARE LOGICAL CHARACTERS UNDER MASK	CLM	RS	C,R	POP370:129
EXCLUSIVE OR	XR	RR	C,R,D	POP370:131
EXCLUSIVE OR	X	RX	C,R	POP370:131
EXCLUSIVE OR CHARACTER	XC	SS	C	POP370:131
EXCLUSIVE OR IMMEDIATE	XI	SI	C,R,D	POP370:131
INSERT CHARACTER	IC	RX	C,R,D	POP370:133
INSERT CHARACTERS UNDER MASK	ICM	RS	C,R	POP370:133
MOVE CHARACTER	MVC	SS	C	POP370:136
MOVE IMMEDIATE	MVI	SI	C,R,D	POP370:136
OR	OR	RR	C,R,D	POP370:140
OR	O	RX	C,R	POP370:140
OR CHARACTER	OC	SS	C	POP370:140
OR IMMEDIATE	OI	SI	C,R,D	POP370:140
STORE CHARACTER	STC	RX	C,R,D	POP370:145
STORE CHARACTERS UNDER MASK	STCM	RS	C,R	POP370:145
TEST AND SET	TS	SI	C,R,D	POP370:148
TEST UNDER MASK	TM	SI	C,R,D	POP370:149
TRANSLATE	TR	SS	C	POP370:149
TRANSLATE AND TEST	TRT	SS	C	POP370:149

Figure 8.2
Logical Operation Instructions

8.3 Branching

Branching instructions alter the execution sequence of a process by generating an address from which the next instruction will be fetched if the branch is successful. The branch address is always computed relative to the M-space from which the branch instruction itself was fetched, so that branching cannot be used for linkage to other modules [Section 5.7]. This local reference convention is also applied to the EXE-

CUTE and LOAD ADDRESS instructions.

The instructions use the RR, RX, and RS formats, in which the first operand field, R1, designates an arithmetic register, except in the conditional branch instructions where it is a mask field which specifies branch conditions. In the RX and RS formats, the B2 field designates an arithmetic register whose contents are added to the displacement D2 and the contents of the arithmetic register designated by X2 (if present), to compute a location value. This value is combined with the identfier of the space from which the instruction was fetched to form the address of the second operand data, or the new value of the process instruction counter on a successful branch. In the RS format, the R3 field designates a pair of arithmetic registers which are used to determine when branching is to occur.

An instruction can designate the same register for address modification and operand specification. The registers designated by an instruction are used first for operand fetch address computation, second for arithmetic computation, and finally for branch address computation. Results, if any, replace the first operand.

Name	Symbol	Type	Class	Description
BRANCH AND LINK	BALR	RR	C,R	
BRANCH AND LINK	BAL	RX	C,R,D	
BRANCH ON CONDITION	BCR	RR	C,R	POP370:124
BRANCH ON CONDITION	BC	RX	C,R,D	POP370:124
BRANCH ON COUNT	BCTR	RR	C,R	POP370:125
BRANCH ON COUNT	BCT	RX	C,R,D	POP370:125
BRANCH ON INDEX HIGH	BXH	RS	C,R	POP370:125
BRANCH ON INDEX LOW OR EQUAL	BXLE	RS	C,R	POP370:125
EXECUTE	EX	RX	C,R,D	POP370:132
LOAD ADDRESS	LA	RX	C,R,D	POP370:134

Figure 8.3
Local Reference Instructions

BRANCH AND LINK

```
BALR  R1,R2  <RR>
BAL   R1,D2(X2,B2)  <RX>
```

The second word of the current process instruction counter [Figure 5.2] is loaded as link information into arithmetic register R1. Subsequently, the LCUR field of the counter is replaced by the branch address.

In the RX format, the second operand location is used as the branch address. In the RR format, the contents of bit positions 8-31 of arithmetic register R2 are used as the branch address. However, when the R2 field is zero, the branch address is set equal to the link address and no branching occurs.

The branch address is always computed before the link information is loaded.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions: None

8.4 Long Operands

Long operands are character strings ranging in length from zero bytes to 16,777,216 bytes. The instructions which manipulate these strings use the RR and RX formats, in which each R-field designates both a general register and a separate arithmetic register as a means of specifying a long operand. The general register, which must have an even value as its designation, is used by itself to form the M-space address of the long operand data field. The corresponding arithmetic register has the next odd value as its implied designation; its contents specify the length of the operand data.

The operands need not reside in the same M-space. However, if they do, and if the fields overlap, the results are consistent with handling the fields one byte at a time, starting at the address of the first byte of each field and proceeding in byte sequence order. Result data fields, if any, replace the first operand.

Name	Symbol	Type	Class	Description
COMPARE LOGICAL LONG	CLCL	RR	C,R,D	
MOVE LONG	MVCL	RR	C,R,D	
TRANSLATE	TRL	RX	C,R,D	New
TRANSLATE AND TEST	TRTL	RX	C,R,D	New

Figure 8.4
Long Operand Instructions

COMPARE LOGICAL LONG

CLCL R1,R2 <RR>

The first operand is compared with the second operand and the result is indicated in the condition code.

The address of the first byte of the first operand is generated from the contents of general register R1, and the address of the first byte of the second operand is generated from the contents of general register R2. The number of bytes in each operand field is specified by the contents of bits 8-31 of the arithmetic registers designated by the values 1+R1 and 1+R2, respectively. The contents of bit positions 0-7 of these registers are ignored.

The comparison is performed with the operands considered as a string of bytes representing unsigned binary integers. The comparison starts at the first byte of each field and proceeds in byte sequential order. The instruction is completed as soon as inequality is detected or the end of the shortest operand is reached. An operand of zero length compares equal with an operand of any length.

At completion, the length registers 1+R1 and 1+R2 are decremented by the number of bytes compared, arithmetic registers R1 and R2 are incremented by the same value, and the condition code is set to reflect the result of the comparison. Bit positions 0-7 of the arithmetic registers remain unchanged.

The instruction is suppressed with a specification exception if either operand field does not contain an even value. It is terminated with an addressing exception if an address is generated during the course of comparison which lies outside a space containing one of the operands.

Process Class: C,R,D

Condition Code:

0	Operands are equal
1	First operand is low
2	First operand is high
3	-

Exceptions:

Access
Addressing
Specification

MOVE LONG

MVCL R1,R2 <RR>

The second operand is placed into the first operand location, except that overlapping of operand locations may affect the final contents of the first operand location.

The address of the first byte of the first operand is generated from the contents of general register R1, and the address of the first byte of the second operand is generated from the contents of general register R2. The number of bytes in each operand field is specified by the contents of bits 8-31 of the arithmetic registers designated by the values 1+R1 and 1+R2, respectively. Bits 0-7 of these registers are ignored.

The movement starts at the first byte of each field and proceeds byte-by-byte in byte sequential order. The instruction is completed when the number of bytes corresponding to the shortest operand field have been transferred. At completion, the length registers 1+R1 and 1+R2 are decremented by the number of bytes moved, and arithmetic registers R1 and R2 are incremented by the same value. Bit positions 0-7 of these registers remain unchanged.

The instruction is suppressed with a specification exception if either operand field does not contain an even value. It is terminated with an addressing exception if an address is generated during the course of movement which lies outside a space containing one of the operands.

Process Class: C,R,D

Condition Code:
0 Fields of equal length
1 First operand shorter than second
2 First operand longer than second
3 -

Exceptions:
Access
Addressing
Specification

TRANSLATE

TRL R1,D2(X2,B2) <RX>

The bytes of the first operand are used as arguments to reference the list located by the second operand address. Each function byte selected from the list replaces the corresponding argument in the first operand.

The address of the first byte of the first operand is generated from the contents of general register R1, and the number of bytes in the field is specified by the contents of bits 8-31 of the arithmetic register designated by 1+R1. The bytes of the first operand are selected one by one for translation, starting at the first byte and proceeding in byte sequential order. Each argument byte is added to the initial second operand address using the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned integer. The sum forms the address of the function byte, which then replaces the original argument byte.

The instruction is completed when the first operand field has been completely replaced. At completion, bits 8-31 of the length register 1+R1 are zero, and arithmetic register R1 is incremented by the length of the operand. Bits 0-7 of these registers remain unchanged. The second operand is not altered unless a field overlap occurs. In that case, the result is the same as if each r4sult byte had been stored immediately after the corresponding function byte was fetched.

The instruction is suppressed with a specification exception if the R1 field does not contain an even value. It is terminated with an addressing exception if an address is generated during the course of translation which lies outside a space containg one of the operands.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions:
Access
Addressing
Specification

TRANSLATE AND TEST

TRTL R1,D2(X2,B2) <RX>

The bytes of the first operand are used as arguments to reference the list located by the second operand address. Each function byte selected from the list is used to determine the continuation of the instruction. If the function byte is zero, the instruction proceeds by fetching and translating the next argument byte. If the function byte is non-zero, the instruction is completed.

The address of the first byte of the first operand is generated from the contents of general register R1, and the number of bytes in the operand is specified by the contents of bits 8-31 of the arithmetic register designated by 1+R1. The bytes of the first operand are selected one by one for translation, starting at the first byte and proceeding in byte sequential order. Each argument byte is added to the intial second operand address using the rules for address arithmetic, with the argument byte treated as an unsigned eight-bit integer. The sum is used as the address of the function byte.

When the instruction is completed, bits 8-31 of the length register 1+R1 are decremented by the number of bytes translated before a non-zero function byte was found, and arithmetic register R1 is incremented by the same value. Bits 0-7 of register R1 are unchanged. If the instruction was completed as a result of finding a non-zero function byte, that byte is inserted into bits 0-7 of register 1+R1, otherwise those bits of the register are unchanged. The condition code is set to reflect the cause of completion.

The instruction is suppressed with a specification exception if the R1 field does not contain an even value. It is terminated with an addressing exception if an address is generated during the course of translation which lies outside a space containing one of the operands.

Process Class: C,R,D

Condition Code:

0	All function bytes zero
1	Non-zero function byte found
2	-
3	-

Exceptions:

- Access
- Addressing
- Specification

8.5 Decimal Feature

A decimal feature instruction set can be installed on any EPSILON system. Decimal instructions provide arithmetic, shifting, and editing operations on data in the S/360 packed and zoned decimal formats [POP360:35]. The instructions use the SS and RX formats. The operand fields in these formats follow the rules described for the logical operation instructions [Section 8.2].

Except for the conversion instructions, CONVERT TO BINARY and CONVERT TO DECIMAL, the decimal feature instructions can be executed only within C-processes.

Name	Symbol	Type	Class	Description
ADD DECIMAL	AP	SS	C	POP370:153
COMPARE DECIMAL	CP	SS	C	POP370:154
CONVERT TO BINARY	CVB	RX	C,R	POP370:130
CONVERT TO DECIMAL	CVD	RX	C,R	POP370:131
DIVIDE DECIMAL	DP	SS	C	POP370:154
EDIT	ED	SS	C	POP370:155
EDIT AND MARK	EDMK	SS	C	POP370:158
MOVE NUMERICS	MVN	SS	C	POP370:138
MOVE WITH OFFSET	MVO	SS	C	POP370:139
MOVE ZONES	MVZ	SS	C	POP370:139
MULTIPLY DECIMAL	MP	SS	C	POP370:158
PACK	PACK	SS	C	POP370:141
SHIFT AND ROUND DECIMAL	SRP	SS	C	POP370:158
SUBTRACT DECIMAL	SP	SS	C	POP370:159
UNPACK	UNPK	SS	C	POP370:150
ZERO AND ADD	ZAP	SS	C	POP370:160

Figure 8.5
Decimal Instructions

8.6 Floating-Point Features

Two floating-point feature instruction sets can be installed on any EPSILON system. The **basic floating-point** instructions operate on data in the S/360 short and long floating-point number formats [POP360:41]. The **extended floating-point** instructions operate on data in the S/370 extended floating-point number format [POP370:162]. The extended floating-point feature can be installed only if the basic feature is also installed.

Floating-point instructions can be executed only within C-processes. If the basic featuure is installed, the state vector of all C-processes is enlarged to include four 64-bit floating-point registers, designated by the numbers 0, 2, 4, and 6. These registers are the referents of the first operand field of all floating-point instructions, and also of the second operand field of the RR format instructions. A specification exception will occur if values other than 0, 2, 4, or 6 are used to designate the registers in the basic instructions, or if values other than 0 or 4 are used in the extended instructions. In the RX format, the B2 field designates a general register whose contents, together with the displacement D2 and the contents of the arithmetic register designated by X2, are combined using the rules of Section 2.8 to form the M-space location of the second operand data.

Number representation, guard digit, an normalization follow the rules and behavior patterns of S/370 [POP370:163-164].

Name	Symbol	Type	Class	Description
ADD NORMALIZED	AER	RR	C	POP370:166
ADD NORMALIZED	AE	RX	C	POP370:166
ADD NORMALIZED	ADR	RR	C	POP370:166
ADD NORMALIZED	AD	RX	C	POP370:166
ADD UNNORMALIZED	AUR	RR	C	POP370:167
ADD UNNORMALIZED	AU	RX	C	POP370:167
ADD UNNORMALIZED	AWR	RR	C	POP370:167
ADD UNNORMALIZED	AW	RX	C	POP370:167
COMPARE	CER	RR	C	POP370:167
COMPARE	CE	RX	C	POP370:167
COMPARE	CDR	RR	C	POP370:167
COMPARE	CD	RX	C	POP370:167
DIVIDE	DER	RR	C	POP370:168
DIVIDE	DE	RX	C	POP370:168
DIVIDE	DDR	RR	C	POP370:168
DIVIDE	DD	RX	C	POP370:168
HALVE	HER	RR	C	POP370:169
HALVE	HDR	RR	C	POP370:169
LOAD	LER	RR	C	POP370:170
LOAD	LE	RX	C	POP370:170
LOAD	LDR	RR	C	POP370:170
LOAD	LD	RX	C	POP370:170
LOAD AND TEST	LTER	RR	C	POP370:170
LOAD AND TEST	LTDR	RR	C	POP370:170
LOAD COMPLEMENT	LCER	RR	C	POP370:170
LOAD COMPLEMENT	LCDR	RR	C	POP370:170
LOAD NEGATIVE	LNER	RR	C	POP370:171
LOAD NEGATIVE	LNDR	RR	C	POP370:171
LOAD POSITIVE	LPER	RR	C	POP370:171
LOAD POSITIVE	LPDR	RR	C	POP370:171
LOAD ROUNDED	LRER	RR	C	POP370:171
LOAD ROUNDED	LRDR	RR	C	POP370:171
MULTIPLY	MER	RR	C	POP370:172
MULTIPLY	ME	RX	C	POP370:172
MULTIPLY	MDR	RR	C	POP370:172
MULTIPLY	MD	RX	C	POP370:172
STORE	STE	RX	C	POP370:173
STORE	STD	RX	C	POP370:173
SUBTRACT NORMALIZED	SER	RR	C	POP370:173
SUBTRACT NORMALIZED	SE	RX	C	POP370:173
SUBTRACT NORMALIZED	SDR	RR	C	POP370:173
SUBTRACT NORMALIZED	SD	RX	C	POP370:173
SUBTRACT UNNORMALOZED	SUR	RR	C	POP370:174
SUBTRACT UNNORMALIZED	SU	RX	C	POP370:174
SUBTRACT UNNORMALIZED	SDR	RR	C	POP370:174
SUBTRACT UNNORMALIZED	SD	RX	C	POP370:174

Figure 8.6
Floating-Point Instructions

Name	Symbol	Type	Class	Description
ADD NORMALIZED	AXR	RR	C	POP370:166
MULTIPLY	MXDR	RR	C	POP370:172
MULTIPLY	MXD	RX	C	POP370:172
MULTIPLY	MXR	RR	C	POP370:172
SUBTRACT NORMALIZED	SXR	RR	C	POP370:174

Figure 8.7
Extended Floating-Point Instructions

9.0 SERVICE PROCESSES

Service processes arise from process models built-in to the system and connected to process sources triggered by certain specified system events. They provide a means to respond to these events with a range of procedures, from simple to complex, not practical to obtain by adjusting parameters of closed functions.

9.1 The System Clock

A real-time clock is included as an integral part of every EPSILON system. The clock is a 64-bit binary counter whose value is an unsigned fixed-point binary number, with binary point located between bits 51 and 52. The zero value represents 1 January 1975, 0 hrs Greenwich Mean Time, which is the epoch standard for all EPSILON systems. Other values of the clock represent microseconds and fractions of a microsecond of time elapsed since the epoch. The actual resolution of the clock may be higher or lower than one microsecond, depending on the model of EPSILON system, but in all models bit positions are incremented at such a frequency that the rate of advancement is the same as if a 1 were added to bit position 51 every microsecond. The incrementing behavior and format of the clock are therefore the same as a System/370 time-of-day clock [POP370:42].

The clock is set during system initialization to a value which is computed by subtracting the epoch from the calendar date and time of initialization. Once set, it runs continually until the system is powered down or re-initialized. Other system activities and events do not affect it, and there are no instructions by which a process can alter its value. Consequently, clock values are consistent for all processes in a system, and provide a consistent means of recording transaction times or specifying when activities are to be initiated. For this purpose, any process can execute a STORE CLOCK instruction (STCK), which stores the current value of the system clock into a double-word located by the instruction operand.

Because of epoch standardization, clock values are also basically consistent between EPSILON systems and between initializations of any particular system. However, unless the clock is tied to an external time-synchronizing signal (e.g. WWV) by the clock synchronization feature, inaccuracies of a second or more in setting its value to true external time are likely to occur because of operator reaction-time delays. These could introduce some inconsistencies between systems. In any event, a clock is always consistent within a single system, and as long as it is running always records elapsed time accurately to within the incrementing resolution.

9.2 Time Events

The system clock is a source for a family of service processes called **time event** processes. The process model connected to it for this purpose is an R-process model whose RMDB has fields filled with fixed data supplied by the system, and empty fields which must be filled before a process of the family can be initiated. The fixed data fields are those which specify conditions of usage or circumscribe the behavior of processes of the family.

- The bits of field RMFLG [Figure 6.1] are set so that
  - process statistics are col-

lected

- the domain identifier of processes of the family is derived from the process which caused initiation

- the process model is single-instance

- the entry context space is bound to the custody of the model

- the process model is in public custody.

o A null exception module is specified by field RMXMD.

▫ The entry context space is specified at system initialization.

The process model is assigned to public custody in order to allow the remaining fields of the RMDB to be specified by instruction execution. The assignment has no other consequences, as the clock does not have an explicit source identifier, and so cannot be referenced by instructions which apply to ordinary sources. In particular, the process model cannot be modified by CRPM or CRPMI, nor can initiation of a process of the family be requested by SGS. For time event processes, both these functions are combined into the REQUEST TIME EVENT instruction (RTE).

The RTE instruction, which can be executed within any C-process or R-process, is unlike other instructions in that the action requested by the instruction is not carried out until the system clock reaches a specified value. The clock value is supplied in a Time Event Request Block (TERB), which also contains process model data. A TERB must begin on a word boundary and have the format described in figure 9.1.

An RTE instruction will be rejected if the TERB is not located within an ordinary M-space which is in process or family custody of the process within which the instruction is being executed. It will also be rejected if the TIME field of the TERB contains a clock value already attained, if the TEMOD field does not identify a module M-space to which the requesting process has read access, or if the TELOC field does not designate an instruction address within the space. If the request is accepted, the space containing the TERB is removed from custody of the process executing the RTE and place into an internal queue called the **timing queue.** Null pointers are loaded into all pointer registers of the process which contain a pointer to the space, and the reference count of the space is decremented by 1 for each null pointer loaded. The RTE itself is then complete.

Spaces in the timing queue are ordered by the clock value of their TERB, with the smallest value at the head of the queue. When the clock attains the value of the item at the head of the queue, the request associated with the item becomes effective.

▫ The space is removed from the timing queue and its reference count is examined. If the count is not zero, the request is nullified by deleting the space from the system.

▫ If the count is zero, the empty fields of the RMDB, which correspond to the instruction counter fields RMMOD, RMMSK, and RMLOC, are filled from fields TEMOD, TEMSK, and TELOC respectively.

o When the RMDB is completed, the equivalent of an SGS instruction [Section 6.3] is executed. The communication space for the SGS

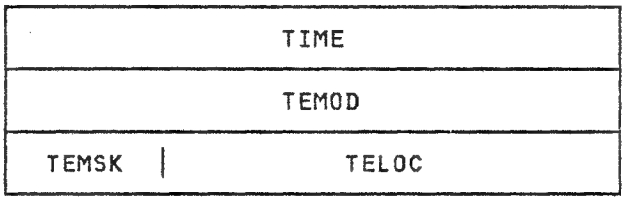

Figure 9.1
Time Event Request Block

Field	Offset	Bytes	Description and Use
TIME	0	8	Clock value at which the time event process is to be initiated.
TEMOD	8	4	A pointer to the module M-space containing the initial instruction sequence to be executed by the process.
TEMSK	12	1	Value of the exception mask for initial entry to the process.
TELOC	13	3	Base address within the M-space identified by field TEMOD of the first instruction of the initial instruction sequence.

is the space which was in the timing queue, and the communication data supplied to the process is the location of the TERB in the space.

Action of the timing queue is inhibited while there is a time event process in existence. This assures the independence of individual time event requests, but as a consequence some requests may not become effective precisely at the specified clock time. Moreover, the actual time of initiation of a time event process after the request is effective depends on the dispatching priority assigned to the clock at system initialization. Therefore, if time event processes are to be used to trigger key events for a system or application, the clock should be assigned a high dispatching priority, and the processes should use as little time for their activity as possible.

9.3 System Exceptions

A system exception is raised during execution of a closed function when an unusual condition is encountered which may be significant to applications, but does not indicate system damage or malfunction. A special class of system exceptions can also be raised by execution of a FORCE SYSTEM EXCEPTION instruction within any process.

An exception or a class of exceptions acts as a source for a family of service processes. The associated process models are defined and connected at system initialization, and as they have no referents cannot be

modified or deleted by ordinary connection instructions, nor can initiation requests be triggered by ordinary request instructions. Because an exception process is intended to provide a means by which action can be taken to note, modify, or remove the exception condition, the processes are supplied with entry data which defines the source and context of the exception. In addition, each process model is granted some special rights of custody and access in order to allow processes of its family to take effective action related to the exception.

There are process models corresponding to the exception sources:

- system overrun
- invalid process model
- forced exception
- statistics collection
- domain end.

The entry data and process model conventions which apply to each exception source are described in the remainder of this chapter.

9.4 System Overrun

A system overrun exception is raised when a computation cycle overrun condition is established [Section 4.7]. The process model connected to the exception is a C-process model whose CMDB contains the following data.

- The bits of field CMFLG [Figure 5.1] are set so that
  - process statistics are collected
  - the domain identifier is fixed as the identifier of the entry context space
  - the entry context space is bound to the custody of the model
  - the process model is in system custody.
- The module space identified by field CMMOD is specified at system initialization, as part of the specification of the computation cycle structure. If the space is the null space, system overruns will be ignored. If the space is non-null, it is bound to the custody of the process model. Fields CMMSK and CMLOC are set to zero.
- The process model is single-instance. The process model name is an internal system name, and is model dependent.
- A null exception module is specified by field CMXMD.
- The entry context space is specified at system initialization.
- There is an associated input queue whose name is the same as the process model name.

When an overrun exception occurs, an M-space is placed into the overrun input queue which describes the content of the computation cycle list at the time of overrun. If there had not been a previous overrun, the queue will be empty; if it is not empty, either an overrun process exists or the queue was not emptied by a previous overrun process. In the latter case, the existing items are deleted from the queue before the new item is entered.

The overrun process model is assigned to a special computation cycle which is higher in precedence than all other computation cycles. Inasmuch as an overrun condition is recognized by dispatching at the expiration of a basic cycle, the initiation request will be honored at once, and initi-

ation will start before any other C-process is assigned a CPU. If field CMMOD of the CMDB identifies an M-space or a B-space with an existing M-space descendent, the overrun process itself will be dispatched before other processes; otherwise, the overrun process will be delayed until an M-space containing its initial instruction sequence is available [Section 5.4].

An overrun process is a regular process which can execute all instructions available to C-processes. In particular, using the STORE CMDB instruction to obtain the names of input queues, it can send messages to any of the C-processes in the computation cycle list. Conventions can therefore be established by which C-processes will accept messages from an overrun process to modify or terminate their activity. If more direct action is desired, the overrun process can itself force termination of other processes. The overrun process model is treated as a system custodian with respect to C-process models, so an overrun process can effectively terminate any C-process by use of the TERM instruction.

Each item placed into the overrun queue consists of a header and a set of computation cycle records, one for each computation cycle. The format of the header is described in figure 9.2; the format of an individual computation cycle record is described in figure 10.3 [Section 10.2].

9.5 Invalid Process Model

An invalid process model exception can be raised either while attempting to initiate a process or to handle a process exception. The exception is raised during process exception handling, for a process of any process class, if the current exception module [Section 5.13] has been deleted from the system, or if the process does not have read access to the space. The exception is raised during initiation

- for a C-process model if initial or exception module data is not valid: field CMMOD must identify either the null space or a module space for which field CMLOC designates instruction location within the space, field CMXMD must identify either the null space or a module space, and field CMCTX must identify either the null space or an ordinary space
- for an R-process model if the M-space identified by field RMMOD has been deleted from the system
- for a D-process model if the M-space identified by field DMMOD has been deleted
- for a model of any class if processes of the family do not have read access to the space containing the initial instruction sequence.

The process model connected to the exception is an R-process model whose RMDB contains the following data.

- The bits of field RMFLG are set so that
  - process statistics are collected
  - the domain identifier for processes of the family is derived from the process which caused initiation
  - the process model is multi-instance
  - the entry context space is bound to the custody of the model

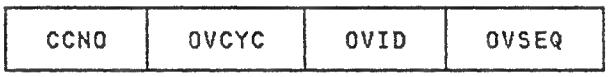

Figure 9.2
Overrun Header Word

Field	Offset	Bytes	Description and Use
CCNO	0	1	Number of computation cycles in the system.
OVCYC	1	1	Overrun cycle indicator [Section 4.7].
OVID	2	1	Identifier of the computation cycle for which this overrun was established.
OVSEQ	3	1	Sequence number of the overrun computation cycle.

- the process model is in system custody.

- The module M-space identified by field RMMOD is specified at system initialization, and bound to to the custody of the model. Fields RMLOC and RMMSK are set to zero.

- A null exception module is specified by field RMXMD.

- The entry context space is specified at system initialization.

When an invalid process model exception occurs, an M-space large enough to contain the process model definition block is allocated, the definition data is copied into it, and the equivalent of an SGS instruction is executed. The location of the process model data becomes the communication data for the exception process initiated as a result of the signal. The source of the exception is supplied in the initial state vector, replacing standard information not applicable to an exception process.

- The condition code describes the class of process which caused the exception. It is zero for a C-process, 1 for an R-process, and 2 for a D-process.

- Arithmetic register 1 contains the name of the process model, or the source identifier, or the device identifier, as appropriate for the process class.

An invalid process model exception process is treated as a system custodian with respect to process models of all process classes. It can therefore modify the data in the entry context space and execute a DCPM, CRPM, or CDPM instruction to remove the defect in the process model, or to delete the model from the system. If an exception process takes no action with respect to the offending process model, the effect on system behavior depends on the cause of the exception and the class of the model.

- If the exception resulted from deletion of an exception module, the process which triggered the exception remains in existence

and is treated as if it had no exception module current. The system exception continues to be raised whenever a process exception occurs for a member of the same family. Hence, in this case the effect is simply an increase in the total system activity generated by process exceptions.

- If the exception is raised during initiation, the initiation attempt is abondoned and the initiation request is deleted from the system without initiating a process. If the request was triggered by entry of a space into an input queue of a C-process model, the space remains in the queue, but does not inhibit further initiation requests by the queue. In such a case, the effect may be indefinite accumulation of spaces into the queue or queues associated with the invalid process model.

To forestall diversion of storage resource into spaces which cannot be used, the system will force deletion of any process model for which a total of 256 invalid process model exceptions are raised while attempting initiation.

9.6 Forced Exception

A system exception is forced by execution of a FORCE SYSTEM EXCEPTION instruction (FSX) within a process of any process class. The process model connected to the exception is an R-process model whose RMDB contains the following data.

- The bits of field RMFLG are set so that
  - process statistics are collected
  - the domain identifier is derived from the process which caused initiation
  - the process model is multi-instance
  - the entry context space is bound to the custody of the model
  - the process model is in system custody.
- The module M-space identified by field RMMOD is specified at system initialization, and bound to the custody of the model. Fields RMLOC and RMMSK are set to zero.
- A null exception module is specified by field RMXMD.
- The entry context space is specified at system initialization.

FSX is similar to SGS in that it transmits communication data to the process initiated in response to the signal. However, as the reason for forcing an exception may well be a condition which inhibits continuation of a process and which the process itself cannot rectify or bypass (e.g. space for critical data cannot be allocated), FSX has a normal behavior which differs somewhat from that of SGS.

- The communication context is expected to be a number, not a location in some M-space. Unless the 'space location' option is selected, the context is generated by combining the null pointer with the arithmetic register designated by the instruction operand.
- If the 'continue activity' option is not selected, a return is made to dispatching to suspend the process within which the instruction is being executed and switch the CPU to another. The

suspended process will be returned to ready dispatching condition upon termination of the exception process initiated in response to the signal.

The dispatching priority of the forced exception process model is assigned at system intitialization. If an R-process whose dispatching priority is higher than the assigned priority is suspended as a result of executing FSX, the corresponding exception process is promoted to a priority higher than the suspended process. A forced exception process is therfore always of higher dispatching priority than any process suspended in its favor.

9.7 Statistics Collection

A statistics collection exception is raised on overflow of a statistics collection counter, or when a queue or process model with active statistics collection is deleted from the system [Section 10.3]. The process model connected to the exception is a C-process model which is assigned to a computation cyle selected at system initialization. The CMDB for the model contains the following data.

- The bits of field CMFLG are set so that
  - process statistics are collected
  - the domain identifier for processes of the family is fixed as the ientifier of the entry context space
  - the entry context space is bound to the custody of the model
  - the process model is in system custody.
- The module space identified by field CMMOD is specified at system initialization. If the space is null, the statistical data collected will be discarded. If the space is not null, it is bound to the custody of the model. Fields CMMSK and CMLOC are set to zero.
- The process model is single-instance. The process model name is an internal system name, and is model dependent.
- A null exception module is specified by field CMXMD.
- The entry context space is specified at system initialization.
- There is an associated input queue whose name is the same as the process model name.

When a statistics collection exception occurs an M-space containing a record of accumulated data is placed into the input queue associated with the model. The format of the record in the space is described in Section 10.5. Statistics collection exception processes are regular processes which can use any of the instructions available to C-processes to dispose of the accumulated data.

9.8 Domain End

A domain end exception is raised when a domain statistics counter overflows, or when a domain end occurs because the membership count of a domain goes to zero [Section 3.5]. The process model connected to the exception is a C-process model which is assigned to a computation cyle selected at system initialization. The CMDB for the model contains the following data.

- The bits of field CMFLG are set so that

- process statistics are collected
- the domain identifier for processes of the family is fixed as the ientifier of the entry context space
- the entry context space is bound to the custody of the model
- the process model is in system custody.

o The module space identified by field CMMOD is specified at system initialization. If the space is null, domain end will be ignored. If the space is not null, it is bound to the custody of the model. Fields CMMSK and CMLOC are set to zero.

o The process model is single-instance. The process model name is an internal system name, and is model dependent.

o A null exception module is specified by field CMXMD.

o The entry context space is specified at system initialization.

o There is an associated input queue whose name is the same as the process model name.

When a domain exception occurs an M-space containing the resource usage statistics accumulated for the domain is placed into the input queue. The record in the space is a special form of statistics record. Its format is described in Section 10.6. Domain end exception processes are regular processes which can execute all instructions available to C-processes.

9.9 Instruction Descriptions

REQUEST TIME EVENT

RTE D1(B1) <SI>

The instruction is terminated with condition code 2 if the operand does not designate a location in an ordinary M-space which is in private or family custody of the process within which the instruction is being executed. It is suppressed with a specification exception if the location is not on a word boundary.

The TERB located by the operand is examined for validity. If field TEMOD does not identify a module M-space to which the requesting process has read access, or if field TELOC does not designate an instruction address, the instruction is terminated with condition code 3. It is terminated with condition code 1 if the value contained in field TIME is smaller than the current value of the system clock.

If the TERB is valid, the space identified by the operand is removed from custody of the process executing the instruction and inserted into the timing queue at a point corresponding to the clock value contained in field TIME. The reference count of the space is decremented by 1. A null pointer is loaded into pointer register B1 and into any other pointer register of the process which contains a pointer to the space, and the reference count is decremented by 1 for each pointer loaded. The instruction is then completed with condi-

tion code zero.

If the reference count of the space is not zero when the time event process corresponding to the TERB is initiated, the process is suppressed and the space is deleted from the system.

Process Class: C,R

Condition Code:

0	Request accepted
1	Time expired
2	Invalid TERB space
3	Invalid TERB data

Exceptions:
Specification

STORE CLOCK

STCK D2(B2) <RS>

The current value of the sytem clock is stored in the double-word located by the second operand. Zeros are supplied in the low order bit positions below the resolution of the clock installed on the system.

The instruction is suppressed with a specification exception if the operand does not define a location on a word boundary.

Process Class: C,R,D

Condition Code: Unchanged

Exceptions:
Specification

FORCE SYSTEM EXCEPTION

FSX M1,R2 <RR>

The conditions are raised which signal a forced system exception.

The contents of general register R2 and bit zero of mask field M1 determine the communication data to be provided to the process initiated as a result of signalling the exception. If bit zero of mask field M1 is zero, the context consists of the null pointer combined with the contents of arithmetic register R2. The contents of the register are not disturbed.

If the bit is 1, the context consists of the full general register. In that case, the instruction is terminated with condition code 3 if pointer register R2 does not identify an ordinary M-space, if the space is in I/O request state, or if the space is not in private or family custody of the process within which the instruction is being executed.

If the instruction is not terminated, a null pointer is loaded into point-

er register R2 and into any other pointer register of the process which contains a pointer to the space. The reference count is decremented by 1 for each pointer loaded. The space is placed into system custody and retained as communication space for the exception process. If the reference count of the space is not zero at the time the exception process is initiated, that process will be suppressed and the space deleted from the system.

Completion of the instruction is determined by bit 1 of mask field M1. If the bit is zero, the process executing the instruction is suspended and control of the CPU assigned to the process is returned to dispatching. The process will be returned to ready dispatching condition when the exception process terminates. The instruction will then be completed with condition code zero if the exception process terminates normally, and with condition code 1 if the process was suppressed.

If bit 1 of mask field M1 is 1, the process executing the instruction continues activity, and the instruction is completed with condition code 2.

Process Class: C,R,D

Condition Code:

0 Exception process completed
1 Exception process suppressed
2 Signal accepted
3 Signal rejected

Exceptions: None

10.0 SYSTEM INQUIRY FACILITIES

System Inquiry facilities provide information describing the configuration, current internal state, and operational history of an EPSILON system. The descriptions are generated from internal tables and lists, or from statistics collected by the system control mechanisms. The basic mechanisms are used just to collect data about individual resources. Other mechanisms record interactions between resources and processes, and so have uses beyond basic data collection.

- Process monitoring mechanisms will record the resource usage of any process, trace its behavior, and raise a process exception if certain specified conditions arise.

- Domain identification mechanisms will accumulate usage statistics by domain identifier. This data, together with the rules for acquisition of domain identifier by processes and spaces, provides a means by which group identification can be used to define, control, and account for units of work which are data-flow analogues of 'jobs' or 'sessions'.

Some statistics are always collected when the collection facility is active, while the collection of others is controlled by instructions which specify what is to be collected and when. Statistics collection degrades sytem performance to an extent which is model-dependent. However, no degradation will occur on any model when the collection facilities are not active.

10.1 Configuration Data

An R-process or C-process can obtain information about the configuration of an EPSILON system using the STORE CONFIGURATION DATA instruction (SCON). Options selected for instruction execution determine whether

- a Configuration Description Block (CDB) is stored

- a computation cycle identifier list is stored

- the information represents installed conditions or current availability.

The format of a CDB is described in figure 10.1. In this figure, if a field or bit description varies according to whether the block represents installed conditions or current availability, the variable part of the description consists of two phrases enclosed in pointed brackets, separated by a vertical line. The first bracketed phrase applies to installed conditions, the second to current availability. If there are no brackets, the field description is invariant.

If the computation cycle identifier option is selected, computation cycle identifiers are stored in successive byte locations, beginning with the computation cycle of smallest period and proceeding in sequence [Section 4.1]. The entire set of identifiers is stored if the installed condition option is selected along with the computation cyle option, while only the identifiers of cycles with an active process are stored if the current availability option is in effect.

Although the SCON instruction options are independently selectable, the location of the stored data

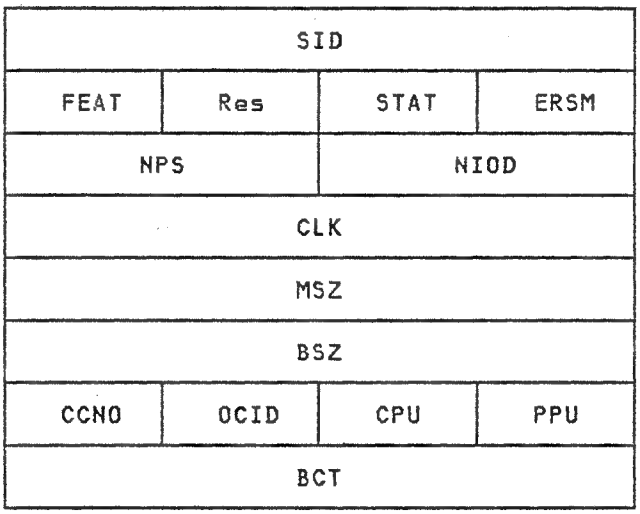

Figure 10.1
Configuration Description Block

Field	Offset	Bytes	Description and Use
SID	0	4	A 32-bit system identifier, assigned at the factory, which distinguishes the system from all other EPSILON systems.
FEAT	4	1	Bits defining the CDB content and system features

Bit	Value	Significance
0	0	Installed conditions are described by this CDB
	1	Current availability is described by this CDB
1	0	The floating-point instruction set is not <installed> | <currently available>
	1	The floating-point instruction set is <installed> | <currently available>
2	0	The decimal instruction set is not <installed> | <currently available>
	1	The decimal instruction set is <installed> | <currently available>
3	0	The clock <is not synchronized to an external timing signal> | <value is valid>
	1	<The clock is synchronized to an external signal> | <clock damage reported but correction not yet applied>
4	0	Process monitoring not <installed> | <currently available>

	1	Process monitoring is <installed> | <currently available>
5	0	Software statistics not <installed> | <currently available>
	1	Software statistics is <installed> | <currently available>
6	-	Reserved
7	0	Normal
	1	Suspension of system operation has been requested.

Field	Offset	Bytes	Description and Use
STAT	6	1	<Initial> | <current> value of the **statistics collection mask.** The bits of this mask determine the conditions under which statistics are collected.

Bit	Value	Significance
0	0	Statistical collection facility <is to be> | <is> inactive
	1	Statistical collection facility <is to be> | <is> active
1	0	Do not collect I/O device statistics
	1	I/O device statistics are to be collected
2	0	Do not collect R-process source statistics
	1	R-process source statistics are to be collected
3	0	Do not collect queue statistics
	1	Queue statistics are to be collected
4	0	Do not collect process statistics
	1	Process statistics are to be collected
5	0	Do not collect domain statistics
	1	Domain statistics are to be collected
6	0	Do not collect software statistics
	1	Software statistics are to be collected
7	0	The value of this mask can be changed
	1	The value of this mask cannot be changed

Field	Offset	Bytes	Description and Use
ERSM	7	1	<initial> | <current> value of the **error signal mask.** The bits of this mask, which determine the conditions under which error signals are effective, are described in Section 11.4.

NPS	8	2	Number of process sources <installed on the system> | <with process models connected>.
NIOD	10	2	Number of I/O devices <installed on the system> | <with process models connected>.
CLK	12	4	High-order 32 bits of the clock value when <the system was last initialized> | <this CDB was stored>.
MSZ	16	4	Number of bytes of M-storage <installed on the system> | <not currently allocated>.
BSZ	20	4	Number of kilobytes of B-storage <installed on the system> | <not currently allocated>.
CCNO	24	1	Number of computation cycles <installed on the system> | <with an active process>.
OCID	25	1	<Overrun cycle indicator [Section 4.7]> | <identifier of the computation cycle for which an overrun was last established>.
CPU	26	1	Number of CPU <installed on the system> | <currently available>.
PPU	27	1	Number of PPU <installed on the system> | <currently available>.
BCT	28	4	Basic cycle time in microseconds.

---

is affected by the combination selected. If both a CDB and a computation cycle list are stored, the list begins at the first location following the CDB. If only a list is stored, it begins at the location designated by the instruction operand.

10.2 Current State Data

C-processes and R-processes have access to process model data by means of the SCMDB, SRCE, and STDS instructions. A C-process can obtain additional current internal state data by use of

- the STORE PROCESS MODEL LIST instruction (SPML), which stores the names of all C-process models in the system
- the STORE DOMAIN LIST instruction (SDOL), which stores the names and identifiers of all domains in the system
- the STORE QUEUE STATUS instruction (SQS), which stores a block describing a specified queue, and
- the STORE COMPUTATION CYCLE RECORD instruction (SCCR), which stores a block describing a specified computation cycle.

The SPML instruction stores the names of C-process models currently in the system, in successive word locations starting at a given location. The

number of words to be stored is specified in the instruction, and a count of the number of names not stored is returned at instruction completion. The names are stored in arbitrary order. The behavior of the SDOL instruction is the same as that of the SPML instruction, except that word pairs are stored, the first word containing the name of a domain and the second word containing its identifier.

The SQS instruction stores a Queue Status Record (QSR), in the format described in figure 10.2, for the queue whose q.ix is specified in the instruction. The C-process within which the instruction is executed need not be a member of the family having custody of the queue.

The SCCR instruction stores a Computation Cycle Status Record (CCSR), in the format described in figure 10.3, for the computation cycle whose identifier is specified in the instruction. The selection routine code contained in field SRT indicates the source of the routine as well as its specific type. Code values 0-127 are reserved for selection routines supplied with EPSILON systems or available as factory installed options. The values for the standard routines [Section 4.4] are:

0 = finite quential
1 = infinite sequential
2 = multiplexed.

Code values of 128-255 are used for custom routines, and routines available only for field installation.

10.3 Collection of Statistics

Statistics are collected in EPSILON systems by incrementing arithmetic counters when an event of statistical significance occurs. A set of standard statistical counters is formed by microcode during operation of all systems. The set varies in size and content as demands for statistics fluctuate. Some of these counters are incremented automatically by microcode; the others are incremented by execution of the COLLECT STATISTICS instruction (CSTAT). A software statistics feature, which can be installed on any model, provides the ability to define counters by instruction execution, and to display the value of counters at any time.

A counter is 8, 16, 32, or 64 bits in length, depending on the kind of data it accumulates. Counters are organized into groups which accumulate data associated with some entity recognized by the system. The type of entity with which a group of standard counters can be associated is

- the system,
- an I/O device,
- an R-process source,
- a queue,
- a process family, or
- a domain.

A software counter group accumulates data collected by some set of processes; the data may or may not be related by other associations.

Counter groups consist of a set of non-addressable counters, which are incremented by microcode, and a set of addressable counters, which are incremented by the CSTAT instruction. There can be a maximum of fifteen addressable counters in any group, with no more than eight 8-bit counters, four 16-bit counters, two 32-bit counters, and one 64-bit counter. The composition of a group is determined by the type of entity with which the group is associated. The system activity group contains only non-addressable counters, a software group contains only adressable counters, while the groups associated

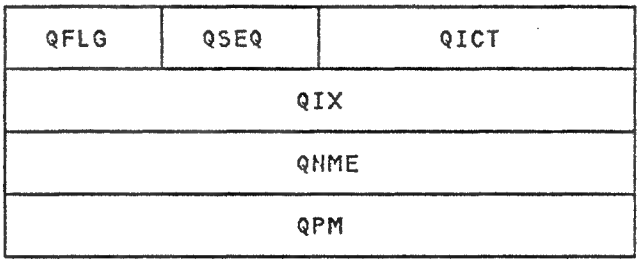

Figure 10.2
Queue Status Record

Field	Offset	Bytes	Description and Use
QFLG	0	1	Bits defining the type and status of the queue.

Bit	Value	Significance
0	0	This is an input queue
	1	This is a public queue
1	0	Normal
	1	This queue has been primed to trigger initiation (input queue only)
2	0	Normal
	1	A process is in queue wait for this queue
3	0	Normal
	1	The value in field QICT is not a true value
4-7	-	Reserved

Field	Offset	Bytes	Description and Use
QSEQ	1	1	Contains the precedence sequence number of an input queue, and zero for a public queue. High precedence corresponds to low number.
QICT	2	2	Count of the number of items in the queue, modulo 16,384. Bit 3 of field QFLG is set to 1 if the true count exceeds 16,383.
QIX	4	4	Q.ix of the queue for which this record was stored.
QNME	8	4	Name of the queue for which this record was stored.
QPM	12	4	Contains the name of the associated C-process model for an input queue, and the name of the custodian process model for a public queue.

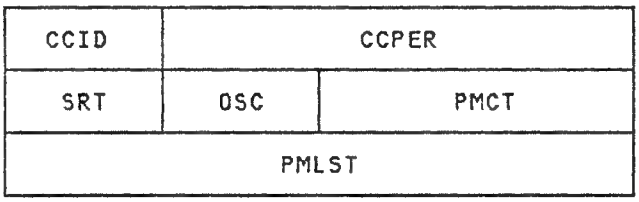

Figure 10.3
Computation Cycle Status Record

Field	Offset	Bytes	Description and Use
CCID	0	1	Identifier of the computation cycle corresponding to the record.
CCPER	1	3	Period of the cycle [Section 4.1].
SRT	4	1	Type of selection rountine in use by the cycle.
OSC	5	1	Overrun sequence count for the cycle [Section 4.7].
PMCT	6	2	Number of process models with a process active in the cycle.
PMLST	8	Var	Names of the process models, listed in arbitrary order; the number of entries in the list is equal to the value of field PMCT.

---

with other types contain both kinds of counters.

The counter affected by a CSTAT instruction is addressed by first selecting a group, and then designating a particular counter within that group. Group selection is accomplished by applying modal conventions to reference a particular group within a designated type of group.

- Group type is designated by a number between 0 and 7:

  0 = system
  1 = I/O device
  2 = R-process source
  3 = queue
  4 = process
  5 = process family
  6 = domain
  7 = software.

- At any given time, a process can address at most one group of any designated type. Types 4, 5, 6, and 7 are treated uniformly for all processes. For type 4, the group selected is the group associated with the process itself. For type 5, it is the group associated with the family of the process. For type 6, it is the group associated with the domain on whose behalf the process is currently acting. For type 7, it is the software group current for the process [Section 10.5].

- Types 1, 2, and 3, which are as-

sociated with process sources, are treated modally. A D-process which designates type 1 refers to the group associated with its own I/O device. An R-process which designates type 2 refers to the group associated with its own source. A C-process which designates type 3 refers to the group associated with its current queue. Any other designation of these types will select a null group of counters.

- Designation of type zero by a process of any class results in selection of a null counter group.

A null group is also selected if the group which normally would be selected does not exist because counter assignment has been suppressed [Section 10.4].

The address of a particular counter within the selected group is always designated by a fixed number, irrespective of whether the counter is actually contained in the group:

1-8 are 8-bit counters
9-12 are 16-bit counters
13-14 are 32-bit counters
15 is the 64-bit counter

while 0 is used to designate the entire set of addressable counters of a group. If the selected group is a null group, or if the designated counter is not contained in the group, the address is taken to refer to a null counter.

To apply these addressing conventions, the RS format is used for the CSTAT instruction. The R3 field specifies the type of group, and the R1 field the particular counter. The R3 field also defines the action to be performed on the counter.

- If the R3 field contains a true type number, the counter is to be incremented. If the field value is a type number plus 8, the counter is to be set to a given value. For example, a 3 in the R3 field specifies that a queue counter is to be incremented, while an 11 in the field specifies that a queue counter is to be loaded with a value.

- The second operand locates the value by which the counter is to incremented or set. The operand length is taken to be equal to the length of the counter.

- If a null counter is selected, the instruction is terminated without taking any action except to set a condition code. The condition is not considered to be an error, and no exception is raised.

The setting of the statistics collection mask, which is displayed in field STAT when a CDB is stored [Figure 10.1], determines whether or not a counter is actually modified when addressed by a CSTAT instruction or by microcode.

- Bit zero of the mask is the master switch. If the bit is zero, the statistical collection mechanisms are inactive. No counters will be modified, either automatically or by instruction execution, and no performance degradation will occur. If the bit is 1, the collection mechanisms are active, as specified by the remaining bits of the mask.

- Bits 1 through 4 are switches for I/O device, R-process source, queue, and process family groups, respectively. Bit 5 controls collection of domain statistics, and bit 6 controls the software counters, if the soft-

ware feature is installed. If a bit is zero, the counters of the type of group it controls are not active. If a bit is 1, counter groups of that type are active, and will be modified if addressed. The system activity counters are not controlled by the mask, as they are always active.

- Bit 7 determines whether or not the value of the mask can be changed. If the bit is zero, the mask can be set to another value; if the bit is 1, the mask value cannot be changed.

The initial value of the statistics collection mask is specified at system initialization. If bit 7 is then zero, the mask can be set to any other value by execution of the SET COLLECTION MASK instruction (SCM) within any C-process or R-process. SCM exchanges mask bytes in the manner of SXM and SBKM, and will continue to do so as long as bit 7 of the new mask remains zero.

10.4 Assignment of Counters

Storage for statistical counters is withdrawn from the M-storage pool when a counter group is to be formed; the storage is returned to the pool when the group is no longer needed. An area of M-storage may be reserved at system initialization exclusively for use by counters. If storage is not available in the reserved area when a counter group is to be formed, an attempt will be made to obtain storage from the general M-storage pool. If the attempt is successful, the storage is allocated to the group, but is not made a part of the reserved area; it will be returned to the general pool when the counter group is deleted.

If storage is not available when a counter group is to be formed, assignment of the group is suppressed, and a null group is substituted in its place. Suppression of counter assignment is a means of limiting the amount of M-storage absorbed by statistics collection. It is not considered to be an error condition. No exception of any kind is raised, nor is the functional behavior of processes affected.

Counter assignment is also suppressed whenever operating conditions indicate that statistics for an entity can never be collected. For example, if bit 7 of the statistics collection mask is 1 and bit 3 is zero, queue statistics cannot be collected unless the system is re-initialized. Consequently, counter assignment is suppressed for queues defined after the mask was set. The rules for counter assignment all follow this principle.

- The system counter group is a unique group which is assigned at system initialization. Assignment is suppressed if at that time bit 7 of the statistics collection mask is 1 and bit zero is zero.

- An I/O device is assigned counters at system initialization. Counter assignment is suppressed if bit 7 of the statistics collection mask is 1 and either bit zero or bit 1 is zero.

- An R-process source is assigned counters at system initialization. Counter assignment is suppressed if bit 7 of the statistics collection mask is 1 and either bit zero or bit 2 is zero.

- A queue is assigned counters when it is defined. Counter assignment is suppressed if bit 7 of the statistics collection mask is 1 and either bit zero or bit 3

is zero. If the queue is an input queue, counter assignment is also suppressed if the C-process model with which it is associated has a CMDB for which bit zero of field CMFLG is zero.

- A process is assigned counters when it is initiated, a process model when it is defined or connected. Counter assignment is suppressed in either case if bit 7 of the statistics collection mask is 1 and either bit zero or bit 4 is zero. Counter assignment is also suppressed if bit zero of the XMFLG field of the XMDB for the model is zero (X being C, R, or D).

Domain counter assignment [Section 10.6] is suppressed by bit 5 of the statistics collection mask, and software counter assignment [Section 10.7] by bit 6.

10.5 Statistics Records

The data collected in the statistical counters is made available when a statistics collection end event occurs. Each such event raises a statistics collection system exception, which causes a statistics collection record (SCR) to be placed into the statistics queue [Section 9.7] and modifies counter values or usage.

- If incrementation would cause a counter of any group to overflow, an overflow SCR is enqueued before the counter is modified. The entire counter group is then reset to zero and normal incrementing proceeds from that point.

- If a queue is deleted from the system by QDEL or during replacement of a C-process model, a queue deletion SCR is enqueued. The counters assigned to the queue are then deleted from the set of standard counters.

- When a process of a family for which statistics are to be collected terminates, a process end SCR is enqueued. The counters assigned to the process are combined into the appropriate counters assigned to the family, and the process counters are then deleted from the set of standard counters.

- When a process model for which family statistics are to be collected is deleted from the system permanently, rather than as an intermediate step during replacement, a process model deletion SCR is enqueued. The counters assigned to the model when it was defined or connected are then deleted from the set of standard counters. When a C-process model with input queues is deleted, the SCR for the queues are placed into the queue following the SCR for the model.

I/O device, R-process source, and system SCR can be enqueued only on counter overflow unless the software statistics feature is installed on the system. Domain statistics are not available through the statistics queue. They are placed into the domain queue [Section 9.8] when a domain system exception is raised.

The general form of an SCR is described in figure 10.4. The counters displayed in the SDTA field of a system SCR are all incremented by microcode whenever an appropriate statistics event occurs, provided the master switch bit of the statistics collection mask is not zero. The format of this SDTA field is described in figure 10.5.

An I/O device is assigned two non-addressable counters, two 16-bit ad-

dressable counters, and eight 8-bit addressable counters at system initialization, if counter assignment is not suppressed at that time. These counters are displayed in the SDTA field of an I/O device SCR, in the format described in figure 10.6. In that figure, the addressable counters have names of the form ACn, where n is an integer whose value is the same as the address value by which the counter is designated. This convention is employed whenever addressable counters are referenced.

An R-process source is assigned two non-addressable counters, two 16-bit addressable counters, and four 8-bit addressable counters at system initialization, if counter assignment is not suppressed at that time. These counters are displayed in the SDTA field of an R-process SCR, in the format described in figure 10.7.

A queue is assigned three non-addressable counters, two 16-bit addressable counters, and one 64-bit addressable counter when it is defined, if counter assignment is not suppressed at that time. These counters are displayed in the SDTA field of a queue SCR, in the format described in figure 10.8.

A process is assigned eight non-addressable counters, one 32-bit addressable counter, two 16-bit addressable counters, and three 8-bit addressable counters, if counter assignment is not suppressed when the process is initiated. These counters are displayed in the SDTA field of a process SCR, in the format described in figure 10.9.

A process model is assigned six non-addressable counters, one 32-bit addressable counter, two 16-bit addressable counters, and three 8-bit addressable counters, if counter assignment is not suppressed when the model is defined or connected. These counters are displayed in the SDTA field of a process family SCR, in the format described in figure 10.10.

Counter assignments and format of a domain statistics record are discussed in Section 10.6. Software statistics formats are described in Section 10.7.

10.6 Domain Statistics

When a domain is formed, it is assigned five non-addressable counters, four 8-bit addressable counters, two 16-bit addressable counters, and one 32-bit addressable counter, provided counter assignment is not suppressed at that time. Counter assignment will be suppressed if bit 7 of the statistics collection mask is 1 and either bit zero or bit 5 is zero. The counters are displayed in the SDTA field of a domain SCR, in the format described in figure 10.11.

A domain statistics record is placed into the domain system exception queue if a counter overflows, or if a domain is deleted because its membership count has gone to zero. The purpose of treating domain statistics separately from other statistics is that domain end is noteworthy in itself; it may indicate, for example, a significant point in the data flow of an application. Consequently, a domain end record (field SCDC equal 4) is always generated when the condition occurs, irrespective of whether statistics for the domain were collected. If there are no statistics, the record ends with field CFRM, and field SLEN [Figure 10.4] will be set to reflect the shorter record length.

10.7 Software Statistics

The software statistics feature comprises three instructions which pro-

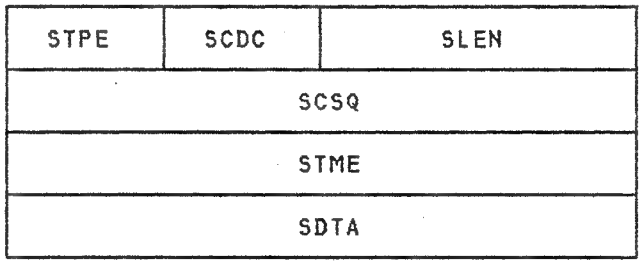

Figure 10.4
Statistics Collection Record

Field	Offset	Bytes	Description and Use
STPE	0	1	An integer between 0 and 7 which designates the type of counter group stored in field SDTA of the record. Values 8-255 are reserved.
SCDC	1	1	An integer which identifies the condition that caused the record to be generated: 0 = counter overflow 1 = process termination 2 = queue deletion 3 = process model deletion 4 = domain end 5 = instruction execution (software feature only) Values 6-255 are reserved.
SLEN	2	2	Length of the record in words.
SCSQ	4	4	Sequence number of the record within the type designated by field STPE. Sequence numbers are set to zero at system initialization, and are reset whenever the corresponding type bit in the statistics collection mask is set to zero.
STME	8	8	System clock time when the record was generated.
SDTA	16	Var	Data accumulated in the counter group for which the record was generated. The format of this field varies with the type designated in field STPE.

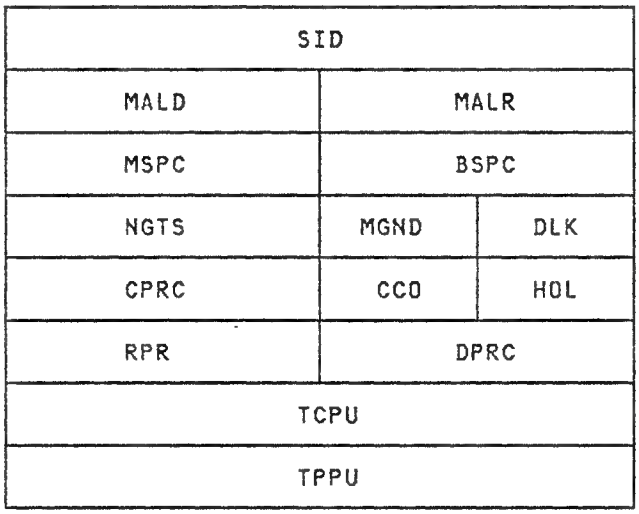

Figure 10.5
System SCR Data

Field	Offset	Bytes	Description and Use
SID	0	4	The system identifier assigned at the factory [Figure 10.1].
MALD	4	2	Number of system malfunctions detected during the period covered by this SCR.
MALR	6	2	Number of system malfunctions reported during the period.
MSPC	8	2	Number of M-spaces allocated during the period.
BSPC	10	2	Number of B-spaces allocated during the period.
NGTS	12	2	Number of access control gates closed during the period.
MGND	14	1	Maximum height of gate promotion during the period.
DLK	15	1	Number of R-process deadlocks detected during the period.
CPRC	16	2	Number of C-processes initiated during the period.
CCO	18	1	Number of computation cyle overruns established during the period.
HOL	19	1	Identifier of the highest level computation cycle for which an overrun was established during the period.

RPRC	20	2	Number of R-processes initiated during the period.
DPRC	22	2	Number of D-processes initiated during the period.
TCPU	24	4	Total time in microseconds for which a CPU was assigned to some C-process or R-process during the period.
TPPU	28	4	Total time in microseconds for which a PPU was assigned to some D-process during the period.

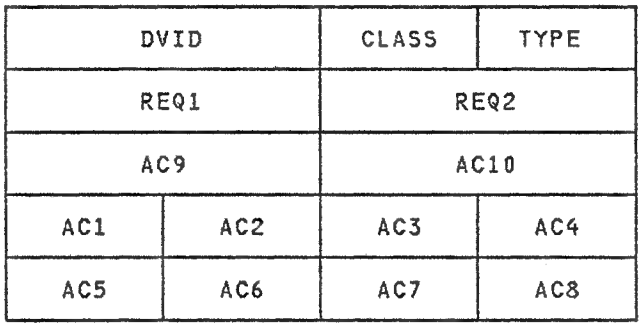

Figure 10.6
I/O Device SCR Data

Field	Offset	Bytes	Description and Use
DVID	0	2	Identifier of the device for which this SCR was generated.
CLASS	2	1	Class code of the device [Figure 7.1].
TYPE	3	1	Type code of the device.
REQ1	4	2	Number of items placed into the request queue for the device during the period covered by this SCR.
REQ2	6	2	Number of items in the queue during the period which were put into I/O request state.
AC9-10	8	2x2	Contents of the 16-bit addressable counters assigned to the device.
AC1-8	12	8x1	Contents of the 8-bit addressable counters assigned to the device.

SRID		Reserved	
TPR		SPR	
AC9		AC10	
AC1	AC2	AC3	AC4

Figure 10.7
R-process Source SCR Data

Field	Offset	Bytes	Description and Use
SRID	0	2	Identifier of the process source for which this SCR was generated.
TPR	4	2	Number of R-processes initiated by the source during the period covered by this SCR.
SPR	6	2	Number of R-processes initiated in the period as a result of an SGS instruction.
AC9-10	8	2x2	Contents of the 16-bit addressable counters assigned to the source.
AC1-4	12	4x1	Contents of the 8-bit addressable counters assigned to the source.

QNME	
QIX	
ICT	IMX
TWTE	
AC9	AC10
AC15	

Figure 10.8
Queue SCR Data

Field	Offset	Bytes	Description and Use
QNME	0	4	Name of the queue for which this SCR was generated.
QIX	4	4	Index of the queue.
ICT	8	2	Number of items placed into the queue during the period covered by this SCR.
IMX	10	2	Maximum number of items in the queue during the period.
TWTE	12	8	Maximum waiting time of an item in the queue during the period. The format of the field is that of the system clock.
AC9-10	20	2x2	Contents of the 16-bit addressable counters assigned to the queue.
AC15	24	8	Contents of the 64-bit addressable counter assigned to the queue.

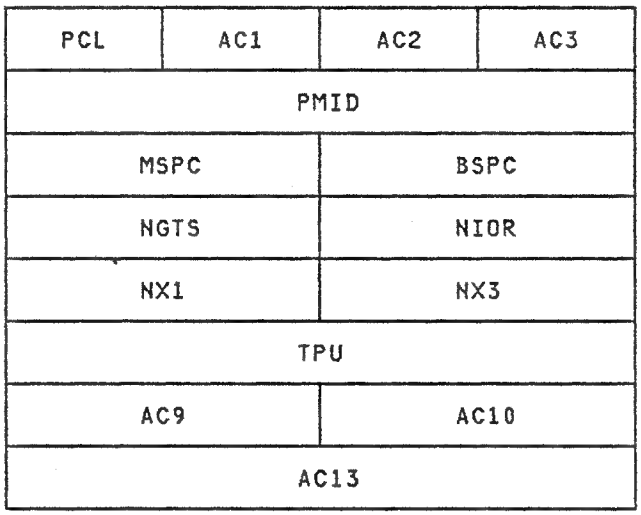

Figure 10.9
Process SCR Data

Field	Offset	Bytes	Description and Use
PCL	0	1	Process class of the process for which this SCR was generated. Field values are zero for a C-process, 1 for an R-process, and 2 for a D-process.

AC1-3	1	3x1	Contents of the 8-bit addressable counters assigned to the process.
PMID	4	4	Process model name if the process is a C-process. If the process is an R-process or D-process, the field contains the process source or device identifier in the low-order bytes.
MSPC	■	2	Number of M-spaces allocated by the process during the period covered by the SCR.
BSPC	10	2	Number of B-spaces allocated by the process during the period.
MGTS	12	2	Number of access control gates closed by the process during the period (zero for ■ D-process).
NIOR	14	2	Number of I/O requests made by the process during the period.
NX1		16	2 Number of class 1 and class 2 process exceptions raised for the process during the period.
NX3	18	2	Number of class 3 and class 4 process exceptions raised for the process during the period.
TPU	20	4	Time in microseconds a processing unit was assigned to the process during the period.
AC9-10	24	2x2	Contents of the 16-bit addressable counters assigned to the process.
AC13	28	4	Contents of the 32-bit addressable counter assigned to the process.

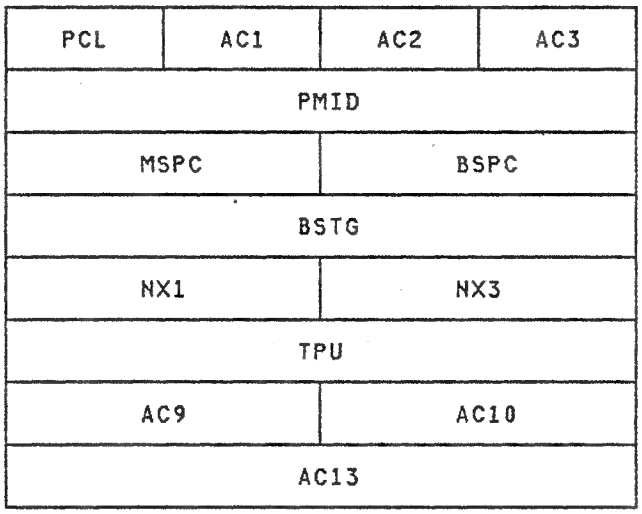

Figure 10.10
Process Family SCR Data

Field	Offset	Bytes	Description and Use
PCL	0	1	Class of the process model for which the SCR was generated [Figure 10.9].
AC1-3	1	3x1	Contents of the 8-bit addressable counters assigned to the model.
PMID	4	4	Process model name for a C-process model. For an R-process model or a D-process model, the field contains the process source or device identifier in the low-order bytes.
MSPC	8	2	Number of M-spaces allocated by processes of the family during the period covered by this SCR.
BSPC	10	2	Number of B-spaces allocated by processes of the family during the period.
BSTG	12	4	Number of kilobytes of B-storage required for B-spaces held in custody of the family during the period.
NX1	16	2	Number of class 1 and class 2 process exceptions raised for processes of the family during the period.
NX3	18	2	Number of class 3 and class 4 process exceptions raised for processes of the family during the period.
TPU	20	4	Time in microseconds processing units were assigned to processes of the family during the period.

AC9-10	24	2x2	Contents of the 16-bit addressable counters assigned to the model.
AC13	28	4	Contents of the 32-bit addressable counter assigned to the model.

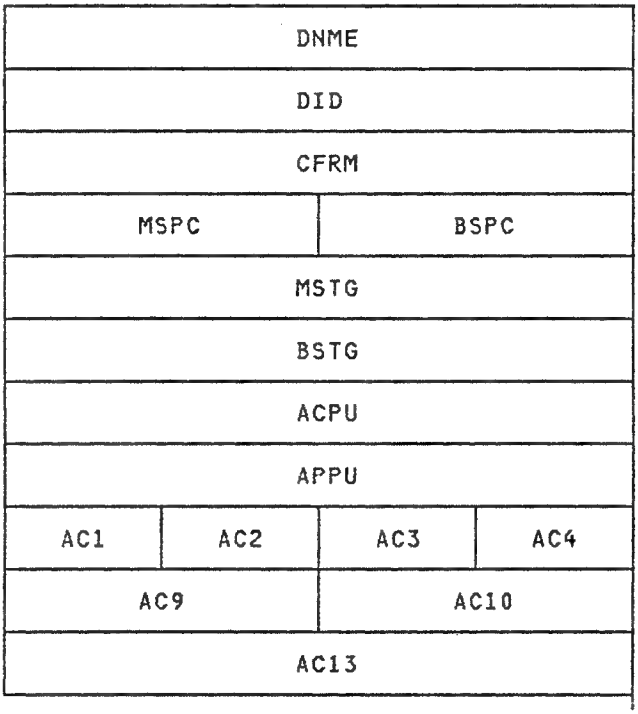

Figure 10.11
Domain SCR Data

Field	Offset	Bytes	Description and Use
DNME	0	4	Name of the domain for which the SCR was generated. The field is zero for the common domain.
DID	4	4	Identifier of the domain.
CFRM	8	8	System clock time when the domain was formed.
MSPC	16	2	Number of M-spaces admitted to the domain during the period covered by this SCR.
BSPC	18	2	Number of B-spaces admitted to the domain during the period.
MSTG	20	4	Number of bytes of M-storage used by members of the domain during the period.

BSTG	24	4	Number of kilobytes of B-storage used by members of the domain during the period.
ACPU	28	4	Time in microseconds a CPU was assigned to some process acting on behalf of the domain during the period.
APPU	32	4	Time in microseconds a PPU was assigned to some D-process acting on behalf of the domain during the period.
AC1-4	36	4x1	Contents of the 8-bit addressable counters assigned to the domain.
AC9-10	40	2x2	Contents of the 16-bit addressable counters assigned to the domain.
AC13	44	4	Contents of the 32-bit addressable counter assigned to the domain.

---

vide processes the ability to define groups of addressable statistical counters, to share the counters with other processes, and to display the value of any group of counters at any time.

- The DEFINE STATISTICAL COUNTER GROUP instruction (DSCG), which can be executed within any C-process or R-process, will define a group of addressable statistical counters, provided M-storage is available and counter assignment is not suppressed. Assignment will be suppressed if bit 7 of the statistics collection mask is 1 and either bit zero or bit 6 is zero.

- If a group is defined, it becomes associated with the defining process as its current software group, and will be selected as such by a CSTAT instruction executed within the process [Section 10.3]. The DSCG instruction also returns a 16-bit identifier for the group, which can be retained by the process or passed to other processes. If a process executes an ACQUIRE STATISTICAL COUNTER GROUP instruction (ASCG) specifying the identifier, the group becomes current for that process also. As in the case of other identifiers, zero is reserved for the identifier of a null group. An ASCG executed with zero as operand will return a process to the condition of having no effective software counter group current.

- The STORE STATISTICAL COUNTER GROUP instruction (SSCG), which can be executed within any process, will store an SCR into a specified location. The counter group for the SCR is selected by the rules used for selecting counter groups for the CSTAT instruction.

The operand of a DSCG instruction is a 16-bit field which defines which of the possible types of addressable counters are to be contained in the group. Bits 1 through 15 correspond to the counters whose addresses within the group are designated by the address values 1 through 15 respec-

tively. If a bit is zero, the corresponding counter will not be included in the group; if the bit is 1, a counter of the appropriate length is assigned to that address.

Bit zero of the operand determines the custody and access of the counter group defined by the instruction. If the bit is zero, the group is private to the process for both custody and access. An ASCG specifying the group executed within any other process will be rejected, and the group is deleted when the process terminates. If the bit is 1, the group is placed into family custody of the defining process; it can then be the legitimate subject of an ASCG instruction. A group in family custody is deleted when the process model is deleted. However, successful execution of an ASCG increments a reference count for the group, just as an LP does for a space. Group deletion will be delayed until the reference count becomes zero.

When the contents of software counters are made available on counter overflow or by execution of an SSCG instruction, the counter group is displayed in the SDTA field of the SCR in the format described in figure 10.12.

<table>
<tr><td colspan="2">CID</td><td colspan="2">CMSK</td></tr>
<tr><td>AC1</td><td>AC2</td><td>AC3</td><td>AC4</td></tr>
<tr><td>AC5</td><td>AC6</td><td>AC7</td><td>AC8</td></tr>
<tr><td colspan="2">AC9</td><td colspan="2">AC10</td></tr>
<tr><td colspan="2">AC11</td><td colspan="2">AC12</td></tr>
<tr><td colspan="4">AC13</td></tr>
<tr><td colspan="4">AC14</td></tr>
<tr><td colspan="4">AC15</td></tr>
</table>

Figure 10.12
Software SCR Data

Field	Offset	Bytes	Description and Use
CID	0	2	Identifier of the software counter group for this SCR.
CMSK	2	2	Mask supplied to the instruction which defined the group. The contents of any of the fields ACn in the record are meaningful only if the bit in this field corresponding to the counter is set to 1.

AC1-8	4	8x1	Contents of the 8-bit addressable counters assigned to the group.
AC9-10	12	4x2	Contents of the 16-bit addressable counters assigned to the group.
AC13-14	20	2x4	Contents of the 32-bit addressable counters assigned to the group.
AC15	28	1	Contents of the 64-bit addressable counter assigned to the group

---

The SSCG instruction will generate an SCR of any type, not just a software SCR. The generated SCR can be stored at a given location or placed into the statistics or domain queues, forcing a system exception of that kind.

- The instruction uses the RX format in which the R1 field specifies the type of group. The particular group within the specified type is selected by the modal conventions used for the CSTAT instruction.

- If the R1 field contains a true type number, the SCR is stored at the location defined by the second operand. If the field value is a type number plus 8, the SCR is placed into the statistics or domain queue, as appropriate.

An SCR generated by an SSCG instruction has format identical to one generated by a statistics event. For a given counter group, the content will also be the same except that field SCDC is set to the value 5 [Figure 10.4]. However, counter values are not altered in any way as a result of generating an SCR by means of an SSCG instruction.

10.8 Process Monitoring

The process monitoring feature allows any process to specify conditions for which the activity of the process is to be monitored. If any of the conditions arise, a monitor trace record is generated, and the process is alerted to the condition by forcing a process exception with the trace information stored in field FRCE of the exception record [Figure 5.4].

A C-process or R-process can be monitored for the following conditions:

- execution of a linkage instruction

- execution of a branch instruction for which branching occurs

- alteration of the contents of designated general registers

- alteration of the contents of a designated M-space

- excessive process time between execution of two monitor control instructions.

A D-process can be monitored for linkage instructions, successful branches, and excessive time, but not for the other conditions. The particular conditions to be monitored are set up by execution of a SET MONITOR CONDITIONS instruction (SMC), which is supplied the monitoring information in a Monitor Conditions Description Block (MCDB). An MCDB

must begin on a word boundary and have the format described in figure 10.13.

When the process monitoring feature is installed, the state vector of every process is enlarged to retain MCDB data, and to include a **monitor mask** which controls whether tracing for specified conditions is active or not. The bits of the mask correspond to the

- linkage,
- successful branch,
- pointer register,
- arithmetic register,
- M-space, and
- process time

conditions, respectively, proceeding from high to low order bits. A bit of the mask must be 1 in order for tracing to be active for the condition to which it corresponds. However, the action of the process monitoring mechanism in relation to a process is governed primarily by bit zero of field PFLG of the PIC.

- If the monitor bit of the PIC is zero, process activity is not monitored at all. No monitor exceptions will be forced, nor will instruction execution time for the process be affected.

- If the monitor bit is 1, instruction execution within the process is monitored for occurrence of the conditions set up by the last SMC instruction executed by the process. Process activity is monitored even if no SMC has yet been executed, so that instruction execution time is always lengthened when the monitor bit is on.

- If the monitor bit is on and a specified condition occurs, a trace record is generated if the monitor mask bit corresponding to the condition is 1. The record is not generated if the mask bit is zero, nor is the process exception forced.

The SMC instruction will also turn on or off the monitor bit of the PIC, and turn on or off all bits of the monitor mask as a group. Consequently, if all conditions are to be treated uniformly, it is possible to set up monitoring conditions and monitoring activity with a single instruction. If conditions are to be treated selectively, individual monitor bits can be altered with the SET MONITOR MASKS instruction (SMM). SMM exchanges both the monitor mask and the monitor bit of the PIC for a mask byte in the instruction.

As process monitoring is considered to be a diagnostic aid which should not interfere with normal process activity, a process monitor exception is suppressed in favor of any other process exceptions raised by an instruction. Moreover, when a monitor exception is forced the new PIC on entry to the exception module is set with monitor bit zero. The exception module is free to turn the monitor bit on, but if a monitor exception is forced during this time it will effectively be ignored [Section 5.13].

The extension of the exception record forced by a monitor exception, corresponding to field FRCE in figure 5.4, has the format described in figure 10.14. The CDATn fields of the extension contain the following information.

- For a linkage exception, CDAT1 contains a pointer to the M-space from which the CALL or RETURN was was executed, and CDAT2 contains the address of the instruction within the space.

- For a branch exception, CDAT1 contains a pointer to the M-space

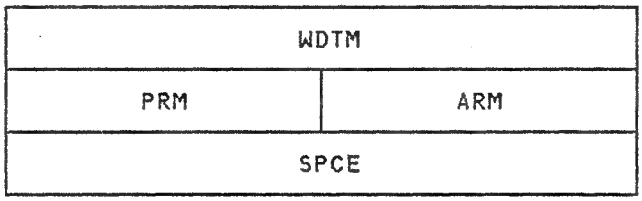

Figure 10.13
Monitor Conditions Description Block

Field	Offset	Bytes	Description and Use
WDTM	0	4	Maximum process time in microseconds between successive executions of an SMC instruction by the process. The time is measured only while a processing unit is assigned to the process.
PRM	4	2	Defines the pointer registers which are to be monitored for alteration. Bits 0 to 15 correspond to registers 0 to 15. A register is to be monitored if its corresponding bit is 1.
ARM	6	2	Defines the arithmetic registers to be monitored for alteration. Bit conventions are the same as for field PRM.
SPCE	8	4	A pointer to an M-space which is to be monitored for alteration.

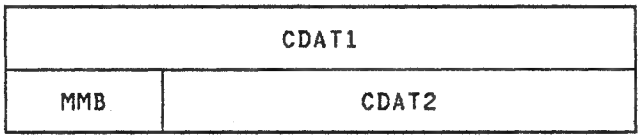

Figure 10.14
Extension of Exception Record
Forced by Monitor Exception

Field	Offset	Bytes	Description and Use
CDAT1	0	4	Contains data pertinent to the condition which caused the exception to be forced.
MMS	4	1	Bits which indicate the condition that caused the exception to be forced. The first six bits correspond to the bits of the monitor mask. The trigger condition is indicated by which one of these bits is set to

			1. The last two bits of the field are zero.
CDAT2	5	3	Contains data pertinent to the condition which caused the exception to be forced.

---

within which the branch instruction is located, and CDAT2 contains the address of the instruction. If the branch was caused by an EXECUTE instruction, the fields locate that instruction.

- For a pointer or arithmetic register exception, CDAT1 contains the value of the register prior to alteration. Field CDAT2 is not significant.
- For an M-space exception, CDAT1 contains a pointer to the space and CDAT2 contains the address within the space of the data which was accessed.
- Neither field has significance for an excessive time exception.

As with any class 3 exception, a monitor trace record is stored after the state vector is updated on completion or termination of the instruction. In each case, therefore, the old and new data values are available to the exception module.

10.9 Instruction Descriptions

STORE CONFIGURATION DATA

SCONS M1,D2(X2,B2) <RX>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. If the instruction is not suppressed, configuration data is stored at the specified location.

The first operand serves as a mask field which selects the options specifiable with the instruction. If the high-order bit of the mask is zero, a CDB is stored at the location; if the bit is 1, no CDB is stored. If a CDB is stored, bit 1 determines its content. If the bit is zero, the CDB describes installed onditions; if the bit is 1, the CDB describes current availability.

If bit 2 of the mask is zero, a computation cycle list is stored; if the bit is 1, no list is stored. Computation cycle identifiers are stored in succesive byte locations, starting at the location immediately following the CDB, or at the location specified by the second operand if no CDB is stored. A complete list of identifiers is stored if the high-order bit of mask field M1 is zero. If the bit is 1, the list contains only the identifiers of those cycles with an active process. In either case, the identifiers are stored in order of decreasing response priority.

Process Class: C,R

Condition Code: Unchanged

Exceptions:
Specification

STORE PROCESS MODEL LIST

SPML R1,D2(X2,B2) <RX>

C-process model names are stored in successive words starting at the second operand location, up to the number of words specified by the value contained in arithmetic register R1. Subsequently the content of the register is replaced by the difference between its original value and the number of C-process models in the system.

Names are stored in arbitrary order. If storing a name would require exceeding the space boundary, the instruction is terminated at that point with condition code 3, and without decrementing register R1. If the instruction is completed, the condition code is set according to the final value of register R1.

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary.

Process Class: C

Condition Code:
0 Difference is zero
1 Differencee is negative
2 Difference is positive
3 Insufficient space allowed

Exceptions:
Specification

STORE DOMAIN LIST

SDOL R1,D2(X2,B2) <RX>

Pairs of domain names and identifiers are stored in successive double-words starting at the second operand location, up to the number of double-words specified by the value contained in arithmetic register R1. Subsequently the content of the register is replaced by the difference between its original value and the number of domains in the system.

The name of the domain is stored in the first word of a pair, followed by the identifier in the second word. The pair corresponding to the common domain is stored first; the remaining pairs are stored in arbitrary order. If storing a pair would require exceeding the space boundary, the instruction is terminated at that point with condition code 3, and without decrementing regi-

ster R1. If the instruction is completed, the condition code is set according to the final value of register R1.

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary.

Process Class: C

Condition Code:

0 Difference is zero
1 Difference is negative
2 Difference is positive
3 Insufficient space allowed

Exceptions:

Specification

STORE QUEUE STATUS

SQS R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is terminated with condition code 1 if arithmetic register R1 does not contain the q.ix of an existing queue.

If the instruction is not suppressed or terminated, a queue status record for the specified queue is stored at the second operand location. The instruction is then completed with condition code zero.

Process Class: C

Condition Code:

0 Status record stored
1 Invalid queue index
2 -
3 -

Exceptions:

Specificattion:

STORE COMPUTATION CYCLE RECORD

SCCR R1,D2(B2) <RS>

The instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary. It is terminated with condition code 1 if arithmetic register R1 does not contain the identifier of an existing computatiion cycle.

If the instruction is not suppressed or terminated, a computation cycle

status record for the specified cycle is stored at the second operand location. The instruction is then completed with condition code zero.

Process Class: C

Condition Code:

0	Status record stored
1	Invalid cycle identifier
2	-
3	-

Exceptions:
Specification

COLLECT STATISTICS

CSTAT I1,I3,D2(B2) <RS>

The instruction is terminated immediately with condition code zero if bit zero of the statistics collection mask is zero. If the bit is 1 the value in a statistical collection counter is incremented or replaced by the value of the integer found at the second operand location.

Immediate fields I1 and I3 specify the counter to be modified and the modification action. If the value of field I3 is between 0 and 7, the counter is to be incremented, otherwise the counter value is to be replaced.

The counter group containing the counter to be modified is selected according to the value of field I3 modulo 8. If the value is zero, a null counter group is selected. If the value is 1, and if bit 1 of the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1, and if the process executing the instruction is a D-process, the counter group selected is the group assigned to the I/O device with which the process is associated. A null group is selected if counter assignment was suppressed for the device, or if the instruction is being executed within a C-process or R-process.

If the value of field I3 modulo 8 is 2, and if bit 2 of the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1 and if the process executing the instruction is an R-process, the counter group selected is the group assigned to the process source of the process. A null group is selected if counter assignment was suppressed for the source, or if the instruction is being executed within a C-process or D-process.

If the value of field I3 modulo 8 is 3, and if bit 3 of the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1 and if the process executing the instruction is a C-process, the counter group selected is the group assigned to the current queue of the process. A null group is selected if the process does not have a queue current, if counter assignment was suppressed for the queue, or if the instruction is being executed within an R-process or D-process.

If the value of field I3 modulo 8 is 4 or 5, and if bit 4 of the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1 and the value is 4, the group selected is the group as-

signed to the process within which the instruction is being executed. If the value is 5 the group selected is the group assigned to the process model for the process. A null group is selected if counter assignment was suppressed for the process or process model.

If the value of field I3 modulo 8 is 6, and if bit 5 of the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1, the group selected is the group assigned to the domain on whose behalf the process executing the instruction is currently acting. A null group is selected if counter assignment was suppressed for the domain.

If the value of field I3 modulo 8 is 7, and if the software statistics feature is not installed on the system or if bit 6 or the statistics collection mask is zero, the instruction is terminated with condition code zero. If the bit is 1, the counter group selected is the software group current for the process within which the instruction is being executed. A null group is selected if there is no such group current for the process.

If a null group is selected, the instruction is terminated with condition code 2. If the group is not null, field I1 is examined to determine the counter within the group which is to be modified. If the field value designates a counter not contained in the selected group, the instruction is terminated with condition code 2. If the designated counter is contained in the group, it is modified by the contents of the field which begins at the second operand location and extends through as many bytes as equal the length of the counter to be modified.

The instruction is completed with condition code 1 if the counter is loaded with a value, or if it can be incremented without overflow. If incrementing would cause overflow, the counter is set to its maximum value, a counter overflow SCR is generated, and a statistics collection system exception is raised by placing the SCR into the statistics collection queue. All counters of the selected group are then reset to zero, and the instruction is completed with condition code 3.

Process Class: C,R,D
Modal

Condition Code:
0 Statistics not being collected
1 Counter modified
2 Null counter selected
3 Counter overflow

Exceptions:
Statistics collection (system)

SET COLLECTION MASK

SCM D1(B1),I2 <SI>

The current value of the statistics collection mask is stored at the location specified by the first operand. The instruction is then terminated with condition code 1 if bit 7 of the mask is 1. If the bit is zero, the mask is set equal to the byte of immediate data in the second operand field, and the in-

struction is completed with condition code zero.

Process Class: C,R

Condition Code:
0 Mask value reset
1 Mask cannot be altered
2 -
3 -

Exceptions: None

DEFINE STATISTICAL COUNTER GROUP

DSCG R1,D2(X2,B2) <RX>

The instruction is suppressed with an operation exception if the software statistics feature is not installed on the system. It is terminated with condition code 1 if bit 7 of the statistics collection mask is 1 and either bit zero or bit 6 is zero.

If the instruction is not suppressed or terminated, an attempt is made to obtain M-storage for a complete set of addressable statistical counters. The instruction is terminated with condition code 2 if storage is not available. If storage is available, counter positions whose addresses correspond to bit positions 1 through 15 of the 16-bit field located by the second operand are marked active if the bit is 1 and inactive if the bit is zero.

If bit zero of the field located by the second operand is zero, the counter group is placed into custody of the process within which the instruction is being executed and marked private. If the bit is 1, the group is placed into custody of the process family.

An identifier for the group is generated and placed into arithmetic register R1, the group is assigned as the current software group of the process, and the instruction is completed with condition code zero.

Process Class: C,R

Condition Code:
0 Counters assigned
1 Assignment suppressed
2 Storage not available
3 -

Exceptions:
Operation

ACQUIRE STATISTICAL COUNTER GROUP

ASCG R1 <RR>

The instruction is suppressed with an operation exception if the software statistics feature is not installed on the system. It is terminated with condition code 2 if arithmetic register R1 does not contain the identifier of a software statistical counter group. If the identifier is valid, the instruction is terminated with condition code 1 if the group is in private custody of a process other than the process within which the instruction is being executed.

If the instruction is not suppressed or terminated, and if the process has a software counter group current, the reference count of that group is decremented by 1. The group referenced by register R1 is then made the current software group of the process, its reference count is incremented by 1, and the instruction is completed with condition code zero.

Process Class: C,R,D

Condition Code:

0	Group acquired
1	Group not available
2	Invalid group identifier
3	-

Exceptions:
Operation

STORE STATISTICAL COUNTER GROUP

SSCG I1,D2(X2,B2) <RX>

The instruction is suppressed with an operation exception if the software statistics feature is not installed on the system. It is suppressed with a specification exception if the second operand does not define a location on a word boundary.

If the instruction is not suppressed, immediate field I1 specifies the type of SCR to be generated and the store action for the SCR. If the value of field I1 is between 0 and 7, the SCR is to be stored at the second operand location, otherwise the SCR is to be placed into the statistics or domain exception queue, as appropriate.

The counter group for the SCR is selected according to the value of field I1 modulo 8. If the value is zero, the instruction is terminated with condition code 2. If the value is 1, and if the process executing the instruction is a D-process, the group selected is the group assigned to the I/O device with which the process is associated. The instruction is terminated with condition code 2 if the device has no counter group assigned, or if the instruction is being executed within a C-process or R-process.

If the value of field I1 modulo 8 is 2, and if the instruction is being ex-

ecuted within an R-process, the counter group selected is the group assigned to the process source for the process. The instruction is terminated with condition code 2 if the source has no counter group assigned, or if the instruction is being executed within a C-process or D-process.

If the value of field I1 modulo 8 is 3, and if the process is being executed within a C-process, the counter group selected is the group assigned to the current queue of the process. The instruction is terminated with condition code 2 if the process has no current queue, if the queue has no group assigned, or if the instruction is being executed within an R-process or D-process.

If the value of field I1 modulo 8 is 4, the counter group selected is the group assigned to the process executing the instruction. If the value is 5, the group selected is the group assigned to the process model for the process. If the value is 6, the group selected is the group assigned to the domain on whose behalf the process is currently acting. If any of these groups is a null group, the instruction is terminated with condition code 2.

If the value of field I1 modulo 8 is 7, the counter group selected is the group assigned as the current software group of the process executing the instruction. The instruction is terminated with condition code 2 if the process has no current software group.

An SCR is generated for the selected group. If the SCR is stored at the second operand location, the instruction is completed with condition code zero. If the SCR is to be enqueued, it is placed into the domain exception queue if the value of field I1 modulo 8 is 6, otherwise it is placed into the statistics exception queue. The instruction is then completed with condition code 1.

Process Class: C,R,D
Modal

Condition Code:
0 SCR stored
1 SCR enqueued
2 Null group selected
3 -

Exceptions:
Operation
Specification
Statistics (system)
Domain (system)

SET MONITOR CONDITIONS

SMC M1,D2(X2,B2) <RX>

The instruction is suppressed with an operation exception if the process monitoring feature is not installed on the system. It is suppressed with a specification exception if the second operand does not define a location on a word boundary.

If the istruction is not suppressed, the information in the MCDB located

by the second operand replaces the monitor conditions current for the process. The entire block is used if the instruction is being executed within a C-process or R-process. For a D-process, only the first word of the block is used.

Mask field M1 is examined for monitor mask action. If bit zero of field M1 is zero, the monitor mask is not disturbed. If the bit is 1, all bits of the monitor mask are set to the value of bit 1 of field M1. If the value of the process time bit of the monitor mask is changed by this action, the process time counter for the process is set to the value in field WDTM of the MCDB located by the second operand.

If bit 2 of field M1 is zero, the instruction is completed with condtition code 2. If the bit is 1, the monitor bit of the PIC is set to the value of bit 3 of field M1. If the new value of the monitor bit is zero, monitor probes for the process are deactivated, and the instruction is completed with condition code zero. If the monitor bit of the PIC is 1, monitor probes are activated, and the instruction is completed with condition code 1.

Process Class: C,R,D
Modal

Condition Code:
0 Monitoring deactivated
1 Monitoring activated
2 Monitoring not changed
3 -

Exceptions:
Operation
Specification

SET MONITOR MASKS

SMM D1(B1),I2 <SI>

The instruction is suppressed with an operation exception if the process monitoring feature is not installed on the system. If the instruction is not suppressed, the value of the monitor bit of the PIC and the current monitor mask of the process are stored in the byte located by the first operand. Bit zero of the byte is set to the value of the monitor bit, bits 1 through 6 with the value of the monitor mask, and bit 7 is set to zero.

The monitor bit of the PIC is then set equal to the value of bit zero of immediate field I2, the bits of the monitor mask of the process are set equal to bits 1 through 6 of field I2, and the instruction completion sequence is entered.

The instruction completion sequence compares the value of bit 6 of the byte stored at the second operand location with the value of the process time bit of the new monitor mask. If the bit values are unequal, the process time counter for the process is set to the value of field WDTM of the current MCDB of the process. The counter is set to its maximum value if there is no MCDB current. If the bit values are equal, the process time counter is not disturbed.

If the new monitor bit of the PIC is 1, monitor probes for the process are

activated, and the instruction is completed with condition code 1. If the bit is zero, the probes are deactivated, and the instruction is completed with condition code zero.

Process Class: C,R,D

Condition Code:

0	Monitoring deactivated
1	Monitoring activated
2	-
3	-

Exceptions:
Operation

11.0 MALFUNCTION DETECTION AND RECOVERY

Mechanisms for the detection and correction of errors caused by malfunction of hardware or microcode vary in form and capability from one model of EPSILON system to another. Some models, for example, employ bit encoding techniques for storage that permit correction of all single-bit failures, while others are able to detect but not correct such failures. All models, however, follow the same basic procedure when an error condition is detected.

- If the detection mechanism is capable of correcting the condition, a local correction is applied. The detection mechanism may then log error correction data, but will take no further action.

- If local correction is not possible, information describing the error condition is transmitted to an error recovery process. For many errors, the recovery process is an internal system process responsible for analysis and correction of a class or type of error.

- If there is no internal recovery process, or if that process cannot effect recovery, a system error signal is raised. The error signal is a process source for a family of service processes which respond to error conditions the system cannot handle. An error signal process may dispose of an error report in any manner, and may trigger orderly system termination if the condition precludes useful continued operation.

A system error signal is raised only for an actual system malfunction, not for programming errors. If an invalid data format or incorrect use of an instruction is detected, the error is reported by signalling a process exception. Conditions which might lead to a deviation from normal for process behavior in general, are reported by signalling a system exception. Errors detected by software are reported by forcing either a process or system exception. This chapter describes the kinds of system malfunction detected, the internal recovery attempted, and the facilities available to error signal processes.

11.1 Hardware Malfunction

Behavioral malfunctions of hardware are detected by encoding redundant bits in storage, or employing redundant circuitry in active components. All **replaceable units** (RU) of EPSILON systems include enough redundancy to determine whether or not a malfunction detected within the unit is the fault of the unit itself. The set of RU for a system is model-dependent, but the CPU, PPU, and Basic Storage Modules (BSM) are always RU.

If an RU detects an error condition it cannot correct, it is said to have sustained **damage**; if the error is corrected the RU has effected **recovery**. An RU which effects recovery may generate a **recovery report** error signal as a means of noting the correction if not itself capable of such action. An RU which sustains damage is removed from the active configuration of the system, and some type of error signal is raised.

- If the RU is replicated, and if it is not the sole remaining active member of its set, a **degradation report** is generated.

- If the RU is not replicated, or

if its peers have all been previously damaged, a **damage report** is generated. If system operation cannot reasonably continue without the damaged RU (e.g. the last CPU failed), **primary** damage is reported, otherwise the damage is **secondary**.

An environmental or operational malfunction, such as low voltage, loss of cooling, or power failure, which causes or may cause behavioral errors, is reported by means of a **warning report**. Warning reports are generated by any RU which detects the condition.

11.2 Invalid System Data

Internal data required for system operation resides in M-storage in the form of control blocks and control block lists. Data values are checked for validity by microcode to the extent practical for a given model. Critical data items in the

computation cycle list,
space allocation list,
process state vectors,
process model definitions,
queue control list,
gate control list, and
device descriptions

are checked on all models. Some form of error signal is raised whenever system data is found to be inconsistent or invalid.

- If the data can be reconstructed, the error is signalled by a recovery report. Reconstruction is possible only on models which preserve redundant system data.

- If a substitute value can be used without affecting the immediate behavior of any process, the error is signalled by a warning report. For example, if queue header data is found to be invalid, the queue can be treated as a null queue without inducing behavioral defects in the attendant or enqueuing processes. However, as in the case of hardware, warning reports signal the certainty of future damage.

- If reconstruction or substitution is not possible, the error is signalled by a damage report. Primary damage is reported if the invalid data prevents normal operation of the system (e.g. invalid gate data prevents R-process dispatching). Secondary damage is reported if the error is limited to a particular process or set of processes.

Reconstruction, substitution, or bypassing of invalid data allows process activity to continue when a recovery report, warning report, or secondary damage report is signalled. However, if primary damage is reported, all process activity is suspended except for any error signal process intiated by the report.

11.3 System Operation Error

Although the microinstruction set is model-dependent, the equivalent of a microprocess exception can occur on all models. If an exception microroutine attributes such an exception to invalid data in the registers or local storage of a CPU or PPU, the exception is treated as a malfunction of that RU [Section 11.1]. RU malfunction is also signalled if a microinstruction sequence private to the CPU or PPU is judged to be at fault.

If a faulty microinstruction sequence is shared by the processing units of a system, a damage report is generated.

- Primary damage is reported if the fault impairs the interpretation

of any instruction standard to the basic, computational, or peripheral instruction sets.

- Secondary damage is reported if the fault impairs the interpretation of an instruction in some feature set, and the feature set is made unavailable. Any process can determine feature availability by storing a configuration description block [Figure 10.1].

Error signals are also raised for unrecoverable internal I/O errors on data transfer between M-storage and B-storage. These will be reported as hardware malfunction if the error is attributed to some RU, and as system operation damage if the microinstruction sequence is judged to be at fault. If the error occurs for an instruction for which a 'space not available' condition code return is possible (e.g. LOAD, SAVE, DCPM), that code is returned to the process executing the instruction. If the error occurs for a CALL instruction executed within a C-process, an access exception is raised when the process is removed from I/O wait dispatching condition.

11.4 Error Signal Mask

The setting of the error signal mask, which is displayed in field ERSM when a configuration description block is stored [Figure 10.1], determines whether or not an error signal actually triggers the error signal process source.

- Bit zero of the mask is the master switch. If the bit is zero, all system error signals are ignored. Error reports are discarded, and no error signal processes will be initiated. If the bit is 1, error signals are effective, subject to the remaining bits of the mask.
- Bits 1 through 5 are switches for primary damage, secondary damage, degradation, warning, and recovery report error signals, respectively. If a bit is zero, signals of the type it controls are ignored. If a bit is 1, signals of that type are effective, and will trigger the error signal process source.
- Bit 6 is the checkpoint control switch. If the bit is zero, a checkpoint request is honored only within an error signal process. If the bit is 1, any R-process can trigger a system checkpoint [Section 12.8].
- Bit 7 determines whether or not the value of the mask can be changed. If the bit is zero, the mask can be set to another value; if the bit is 1, the mask value cannot be changed.

The initial value of the error signal mask is specified at system initialization. If bit 7 is then zero, the mask can be set to another value by execution of the SET ERROR MASK instruction (SRM) within any C-process or D-process.

If an error signal is raised when the master switch bit is off, or when the master switch bit is on but the bit which controls the type of error reported is off, system activity continues if at all possible. However, the system termination procedure is invoked if system damage is so severe that activity cannot continue. System termination will signal an external alarm, attempt to checkpoint system data, and stop all CPU, PPU, and I/O activity [Chapter 12].

11.5 Error Signal Processes

The process model connected to the error signal process source is an R-process model whose RMDB contains

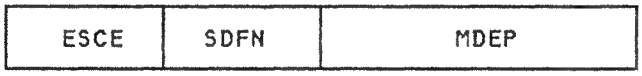

Figure 11.1
Error Report

Field	Offset	Bytes	Description and Use
ESCE	0	1	Bits which define the type and source of the error report

Bit	Value	Significance
0	0	Normal
	1	This is a primary damage report
1	0	Normal
	1	This is a secondary damage report
2	0	Normal
	1	This is a degradation report
3	0	Normal
	1	This is a warning report
4	0	Normal
	1	This is a recovery report
5	0	Normal
	1	This report was generated because of a hardware malfunction
6	0	Normal
	1	This report was generated because of invalid system data
7	0	Normal
	1	This report was generated because of a system operation error.

Field	Offset	Bytes	Description and Use
SDFN	1	1	Bits which define the source which caused the report to be generated. The significance of each bit varies with the type of source. In the following description, numbers in parentheses indicate the source type to which a statement is applicable: (1) indicates hardware malfunction, (2) indicates invalid system data, and (3) indicates system operation error. If a statement has no source type numbers attached, it applies to all source types.

Bit	Value	Significance
0	0	Normal
	1	Reported by ■ CPU (1) Computation cycle list (2) Power abnormality (3)
1	0	Normal
	1	Reported by a PPU (1) Space data or space allocation(2) Power failure (3)
2	0	Normal
	1	BSM affected (1) Process state vector (2) Cooling system failure (3)
3	0	Normal
	1	System clock affected (1) Process model definition block (2) Reserved (3)
4	0	Normal
	1	Queue header or queue control (2) Reserved (1,3)
5	0	Normal
	1	Gate or gate control list (2) Reserved (1,3)
6	0	Normal
	1	Device description (2) Reserved (1,3)
7	0	Normal
	1	A source other than one specifically identified by the other bits

Field	Offset	Bytes	Description and Use
MDEP	2	2	Data which depends on source and model. The field is described in the functional specifications of each EPSILON system model.

---

the following data.

- The bits of field RMFLG are set so that
  - process statistics are not collected
  - the domain identifier for processes of the family is the identifier of the entry context space
  - the process model is

single-instance, and is in system custody.

- The module M-space identified by field RMMOD is specified at system initialization, and bound to the custody of the model; fields RMLOC and RMMSK are set to zero. A null exception module is specified by field RMXMD.

- The entry context space is specified at system initialization.

The error signal process source is assigned a dispatching priority higher than any other R-process source, so that an attempt is made to initiate an error signal process as soon as an error signal becomes effective. The system termination procedure will be invoked if system damage is so severe that a process cannot be initiated. System termination will also be invoked if an error signal process is itself terminated by any means other than execution of an EXIT instruction.

If a process is initiated, the error report becomes the communication data placed into arithmetic register 1 at entry to the process. The format of an error report is described in figure 11.1. An error signal process may respond to the error report with any kind of activity allowed to an R-process. If the process terminates with an EXIT instruction, the implication is that damage is not extensive enough to warrant shutdown of the system. If system operation is to be shutdown, the error signal process can output system restart data with a CHECKPOINT instruction and application or installation restart data using normal I/O operations. The process can also trigger logout of data to be analyzed to determine the cause of failure.

Some diagnostic data is generated for every error signal. If a CPU, PPU, or BSM sustains damage, error logout data is placed into a reserved area of M-storage before the RU is removed from the active configuration of the system. Data may also be placed into the logout area by microcode prior to generation of a damage report, and by execution of a DIAGNOSE instruction.

DIAGNOSE is a special instruction which can be executed only within an error signal process. It will logout diagnostic data and then invoke system termination. No operand is specified, as the type of data to be placed into the logout area is determined by the error report which initiated the process within which the instruction is executed. The specific data logged for a given type of error report is model-dependent, and each model has its own diagnostic microroutine to analyze data stored in the logout area. These routines, which can be executed only when the system is in diagnostic mode, are described in the functional characteristics document for each model.

11.6 Instruction Descriptions

SET ERROR MASK

SRM D1(B1),I2 <SI>

The current value of the error signal mask is stored into the byte located by the first operand. The instruction is then terminated with condition code

1 if bit 7 of the mask is 1. If the bit is zero, the mask is set equal to the byte of immediate data in the second operand field, and the instruction is completed with condition code zero.

Process Class: C,R

Condition Code:
0 Mask value reset
1 Mask value cannot be reset
2 -
3 -

Exceptions: None

DIAGNOSE

DIAGNOSE - <RR>

The instruction is suppressed with an operation exception if the process within which it is being executed is not an error signal process.

If the instruction is not suppressed, error logout data corresponding to the error report being serviced by the process is placed into the diagnostic logout area, and the instruction is completed by invoking the system termination procedure.

Process Class: R

Condition Code: Unchanged

Exceptions:
Operation

12.0 SYSTEM INITIALIZATION

System initialization is the activity of bringing a system into productive operating condition after power is turned on, or after system operation has been terminated for any reason. The particular operating condition achieved is determined by the content of the Initialization Data Table (IDT) supplied as input to the initialization procedure.

An IDT contains all the configuration data, initial space data, process model definition blocks, and system parameter values necessary to specify an operational state corresponding to the desired operating condition. The initialization procedure is a sequence of steps which carries the system from its initial operational state to the state defined by the IDT. Because IDT can be prepared at any time, system initialization is equally capable of producing states which correspond to initial startup following delivery or to a restart from some checkpoint.

12.1 Overview

System initialization is invoked from the operator console by designating an input device with the device-identifier switches and then depressing the initialization key. On some models it is possible to attach an I/O device so that it can transmit a signal which duplicates that of the initialization key. In either case, the initialization signal is effective only if the system has been reset to initial operational state.

Initial state is entered at the end of a power-on sequence, as the final step of system termination, or when the system is reset as a result of depressing the system reset key on the operator console. In the initial state, process models are not connected to any process source, the system data areas are clear, instruction interpretation by CPU and PPU is suspended, and I/O reset has been signalled to all devices attached to the system. Consequently, there is no process activity, and none can be initiated until the initialization procedure has been completed.

When an initialization signal becomes effective, the IDT is loaded into M-storage from the initialization device. If the console switches are set to the null device (identifier zero), the IDT is obtained from an area in B-storage set aside for initialization data. The area is supplied at the factory with a basic startup IDT, and is replaced during each operational period with an IDT which provides continuity of operation from one operational period to the next. If the console switches are set to a real device, the IDT is loaded by a special microprogram sequence executed by a PPU through which the device is attached to the system. If the device is passive (e.g. tape unit or disc), it is assumed to be properly positioned for input; if the device is active (e.g. another computer system), it is expected to be able to transmit IDT records on request. A primary damage error signal is raised if the IDT cannot be loaded without error; the signal will force system termination as an error signal process model is not connected.

Once an IDT has been successfully loaded, initialization progresses through a sequence of phases which correspond to the organization of data within the IDT. An IDT consists of a maximum of nine sections of data, preceeded by a header, and for each section there is an initializa-

tion phase whose responsibility is to convert the data in the section into system data and activity appropriate to the desired operational state. The sections and phases, which for convenience are referred to by the same names, are processed in the following sequence.

- **System parameters.** This section contains configuration description block values [Figure 10.1], M-storage area reservation sizes, and general system data. The initialization phase for the section simply sets parameters to the given values.

- **Dispatching structure.** This section contains R-process source dispatching priority values, and the characteristics of each computation cycle in a descriptive form similar to a computation cycle status record [Figure 10.3], arranged in order of computation cycle precedence. The initialization phase for the section builds the dispatching structure corresponding to this data in the system data area.

- **Space Definition.** This section contains entries which specify the length and content of module and data spaces which are required to define process models, or which must be present in the system prior to initiation of the first regular process. Each entry has a name by which it can be referenced from other sections of the IDT, and may also have an associated domain name. For each entry, the initialization phase for the section allocates a space of the given size, stores the data into it, and assigns the space to the given domain, newly formed if necessary.

- **Service process models.** This section contains the data to complete those process models. References to spaces are either in terms of a name for an entry in the space definition section of the IDT, or in terms of pointers to B-spaces which previously existed in the system. The initialization phase for the section is then able to define and connect the process models to their process sources.

- **R-process models, D-process models, C-process models.** These sections contain entries which are process model definition blocks of the given class. As in the case of the service process models section, space references are either in terms of a name for an entry in the space definition section or are B-space pointers. The initialization phases for these sections define or connect the process models corresponding to the entries.

- **System checkpoint.** This section, which is present only if the IDT was generated by a CHECKPOINT instruction [Section 12.8], contains internal system data saved by the checkpoint; it is the only section of an IDT whose data format is model-dependent. The initialization phase for the section restores the information to the system data area. It may also allocate M-spaces by LOAD instructions applied to B-spaces containing data saved during the checkpoint, and set up state vectors for processes whose activity is to restart from the point of suspension.

- **Application initialization.** This section contains data whose structure and content is not known to initialization. The data may have been created at any time, and in any manner, and attached to the IDT when it was

stored on the initialization device. The section also contains the name of an input queue associated with some C-process model. The initialization phase for the section completes the initialization procedure, and as the final step allocates an M-space, places the section data into it, enqueues the space on the given queue, and releases the system to normal operation. If the process initiated as a result of the enqueue is dispatched in the first computation cycle, it will be able to use the data to set up starting conditions for an application or operating system.

The header of an IDT is the only part which is required to be present. If a data section is missing, the initialization phase for the section will substitute default values for data which is essential to system operation, and will omit all activity related to non-essential data. The default values, which are the same for all systems, need not be the values in the IDT installed in the B-storage initialization area. If an error or inconsistency is noted in the data of any section, initialization will proceed as if the section were missing. A warning report will then be generated when the initialization procedure is completed.

12.2 Initialization Data Table

In order to simplify the task of constructing an IDT, data sections are treated as independent, self-contained units, and may appear in any order. The first byte of each section is an identifier whose value specifies the section content as:

0 = system parameters
1 = dispatching structure
2 = space definition
3 = service process models
4 = R-process models
5 = D-process models
6 = C-process models
7 = system checkpoint
8 = application initialization

If a section is of variable length, the three bytes which follow the identifier byte specify the length. Using the identifier and length values, initialization will search the table, if necessary, to locate a section required for a given phase. The structural ordering of an IDT is not entirely arbitrary, however, as the first item must be a header in the format described in figure 12.1. The SID and CLOK fields of the header are provided as information by the system checkpoint function. A valid IDT may be constructed with arbitrary values for these fields, as they are not examined or used in any way by initialization.

After a system has once been put into productive operation, subsequent initializations may be carried out on the data base preserved in the system from a previous period of operation.

- If field DBI of the header is zero, previous data is to be ignored, and the space allocation list is set up as initially empty. Consequently, space references in the IDT must all be in terms of entries in the space definition section, as there are no pointers initially valid.

- If field DBI is non-zero, the B-spaces, B-space pointers, and B-space allocation list present in the system from the previous period of operation form the data base for initialization. The allocation list is set up by fetching the B-space list resident in B-storage and setting the M-space list empty, as M-spaces cannot be preserved. Space references in the IDT may be entries in the space definition section

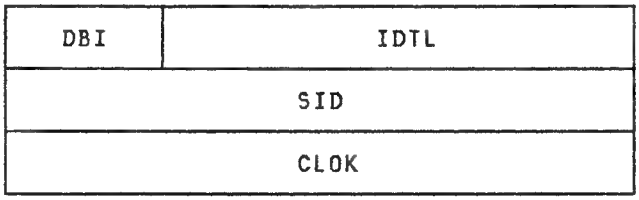

Figure 12.1
Initialization Data Table Header

Field	Offset	Bytes	Description and Use
DBI	0	1	Describes the initial data base. If the field is zero, any data resident in the system is to be ignored. If the field is non-zero, resident B-space data is to be accepted as current.
IDTL	1	3	Combined length in bytes of the header and all data sections in the IDT.
SID	4	4	Identifier of the system.
CLOK	8	8	Time in system clock form when the IDT was constructed or generated.

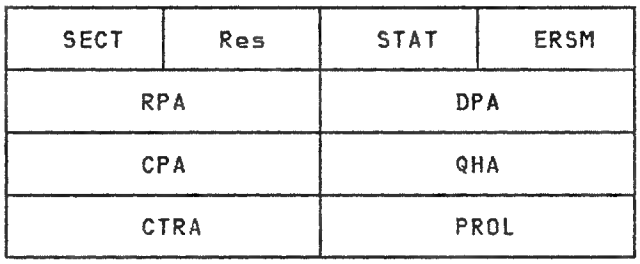

Figure 12.2
Initialization Data Table
System Parameters Section

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value zero, indicating this is the system parameters section of the IDT.
STAT	2	1	Initial value for the statistics collection mask.
ERSM	3	1	Initial value for the error signal mask.

RPA	4	2	Count of the number of R-process sources for which process model and state vector space should be reserved in M-storage.
DPA	6	2	Count of the number of I/O devices for which process model and state vector space should be reserved in M-storage.
CPA	8	2	Number of bytes of M-storage to be reserved for C-process models.
QHA	10	2	Number of bytes of M-storage to be reserved for queue header data and C-process model state vectors.
CTRA	12	2	Number of bytes of M-storage to be reserved for statistical counters.
PROL	12	2	Value to be used for the promotion limit for gates held by R-processes [Section 6.5]. The nominal limit will be used if the field is zero.

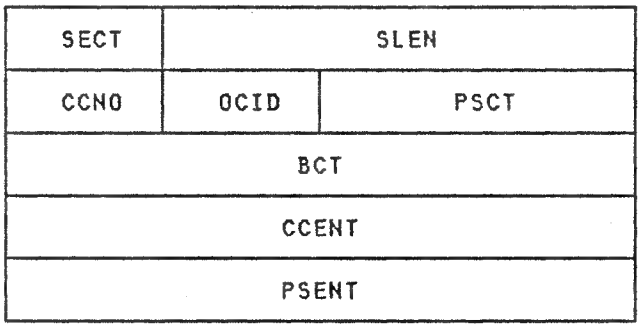

Figure 12.3
Initialization Data Table
Dispatching Structure Section

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value 1, indicating this is the computation cycle structure section of the IDT.
SLEN	1	3	Total length of the section in bytes.
CCNO	4	1	Number of computation cycles to be installed on the system.
OCID	5	1	Overrun cycle indicator.

PSCT	4	2	Number of process source identifiers contained in field PSENT.
BCT	8	8	Time in system clock format which is to be used for the basic cycle.
CCENT	16	Var	Entries specifying the computation cycle characteristics, arranged in order of cycle precedence. The number of entries is equal to the value of field CCNO.
PSENT	Var	Var	Process source identifiers, ordered by relative dispatching priority.

---

or B-space pointers which identify an existing space.

Although an IDT with data base indicator field non-zero is the simplest way of providing continuity from one period of operation to the next, a zero-indicator field IDT can also be used if the space definition section contains entries for all spaces to be preserved. Such an IDT is required, in fact, if the system is to be restored to an operational state not consistent with the state at termination of the previous period, as is the case for a full checkpoint [Section 12.8].

12.3 System Parameters

The system parameters data section of an IDT is fixed in length, with the format described in figure 12.2.

In addition to setup of the parameter values from fields STAT, ERSM, and PROL, the system parameters initialization phase structures the system data area to reflect the space requests, if feasible. If the space requests are judged excessive for the amount of M-storage installed on the system, they are reduced proportionately to satisfactory values; the amount of reduction is model-dependent. If the section is not present in the IDT, the phase substitutes zeros for all field values.

12.4 Dispatching Structure

The dispatching structure section of an IDT is of variable length, with the format describe in figure 12.3.

The dispatching structure initialization phase forms the initial dispatching tables in the system data area. If field PSCT is zero, or if the section is not present in the IDT, the phase assigns factory-installed values as the dispatching priorities of the R-process sources. If the field is not zero, new dispatching priorities are assigned to all sources, starting with the sources whose identifiers are contained in field PSENT. The relative priorities are assigned in the order the sources appear, with the first source receiving the highest priority. Priorities are then assigned to the sources not specified by selecting them in the order of their factory-installed priorities.

Each entry in the computation cycle portion of the section, field CCENT, is a double-word with the format described in figure 12.4.

CCID	CCPER	
SRT	OSC	SRD

Figure 12.4
Initialization Data Table
Computation Cycle Entry

Field	Offset	Bytes	Description and Use
CCID	0	1	Identifier of the computation cycle corresponding to the entry.
CCPER	1	3	Period of the computation cycle.
SRT	4	1	Type number of the selection routine for the cycle.
OSC	5	1	Overrun sequence count for the cycle.
SRD	6	2	Reserved for use as selection routine input data.

In addition to formation of the specified computation cycle structure in the system data area, the initialization phase constructs a special time event request block for the basic cycle. This TERB will be inserted into the time event queue in the final phase of initialization, and will remain in the queue effectively at all times, as the microcode process invoked by the event will request reinsertion of the updated TERB.

If the section is not present in the IDT, the phase will assume the system is to operate with a single computation cycle of indefinite period governed by the infinite sequential selection routine [Section 4.4]. The identifier of the cycle is set to zero. The overrun cycle indicator and overrun sequence count are also set to zero, but their values have no effect on system operation as an overrun cannot occur. The basic cycle time is set to 1.048576 seconds, obtained by inserting a 1 into bit position 31 of the 64-bit clock field.

12.5 Space Definition

The space definition section of an IDT is variable length, with the format described in figure 12.5. The space definition entries in field SDENT of the section are themselves variable in length, with the format described in figure 12.6.

The space definition initialization phase constructs a table of name-pointer pairs, one pair for each entry in the section, which is accessed by the remaining initialization phases whenever a pointer is required for a space referred to by name. An empty table is constructed if the section is not present in the IDT. When the section is present, space definition is carried out in the following way.

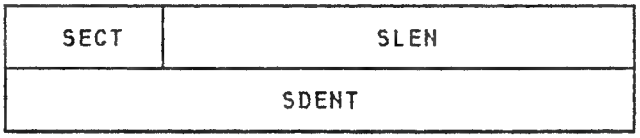

Figure 12.5
Initialization Data Table
Space Definition Section

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value 2, indicating this is the space definition section of the IDT.
SLEN	1	3	Total length of the section in bytes.
SDENT	4	Var	Entries specifying the spaces which are to be defined.

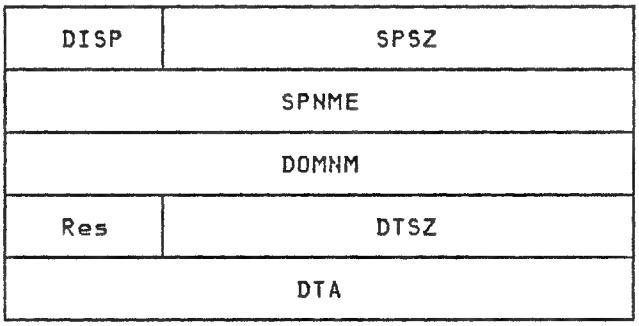

Figure 12.6
Initialization Data Table
Space Definition Entry

Field	Offset	Bytes	Description and Use
DISP	0	1	Bits which describe the initial disposition of the space.

Bit	Value	Significance
0	0	The space is to be an ordinary space
	1	The space is to be a module space
1	0	The space is to remain an M-space
	1	The space is to be converted to a B-space

2	0	Assign the space to the common domain
	1	Assign the space to the domain whose name is in field DOMNM
3	0	Space custody may be transferred
	1	The space is to be bound to system custody
4	0	Read access is to be family
	1	Read access is to be domain or public
5	0	Write access is to be family
	1	Write access is to be domain or public
6	0	Normal
	1	The space pointer is required to be a pointer from a previous operational period
7	-	Reserved

Field	Offset	Bytes	Description and Use
SPSZ	1	3	Number of bytes to be allocated for the space.
SPNME	4	4	Name by which the space can be referenced within any initialization phase.
DOMNM	8	4	Name of a domain to which the space is to be assigned. The significance of this field is controlled by bit 2 of field DISP.
DTSZ	13	3	Number of bytes of data in field DTA.
DTA	16	Var	Data to be stored into the space.

- An M-space is allocated equal in size to the value of field SPSZ of the entry, and the data from field DTA is placed into it. The type of space allocated is controlled by bit zero of field DISP. The space is in system custody, with family (i.e. system) read and write access.
- If bit 2 of field DISP is 1, the equivalent of an ASSIGN instruction referencing the name in field DOMNM is applied to the space, causing a domain of that name to be formed if one did not previously exist. If the domain already exists, the space is simply assigned to it, and the membership count increased by 1. If bit 2 of field DISP is zero, the space remains in the common domain.
- If an access bit of field DISP is 1, and if the space was assigned to a domain other than the common domain, the corresponding access type is set to domain; if the space was assigned to the common domain, the access type is set to public.
- If bit 1 of field DISP is 1, a

B-space is allocated by the equivalent of a SAVE instruction applied to the space. The original space is deleted from the system when the save is successfully completed, and the B-space pointer replaces the M-space pointer in the name-pointer table.

- The space is either bound to system custody or left unbound, as determined by the value of bit 3 of field DISP. If it is not bound, it may be bound to custody of some process model during a subsequent phase of initialization. If so, any access type of family refers to the new custodian.

If the IDT was generated by a checkpoint, the reference name in field SPNME of an entry is actually the pointer assigned to the space at that time, and bit 8 of field DISP is set to 1 for all M-space entries, and for all B-space entries whose pointers must be restored to their previous values. For those entries, the space definition phase supplies the space allocator with the entry name, and if it is consistent with a pointer for the type of space requested, the allocator will assign the name as the pointer. A pointer substitution flag set by the phase will cause an allocation list consistency check to be carried out in the system checkpoint phase [Section 12.8].

12.6 Service Process Models

The service process models data section of an IDT is fixed length, with the format described in figure 12.7. The service process models initialization phase defines or connects the process models whose data is contained in the section. The R-process source dispatching priorities are set to their final values by assigning the specified priorities to the sources of the section, with the error signal source always having the highest priority. The other sources in the system are reduced in priority to accommodate insertion of the section source priorities if necessary. If the section is not present in the IDT

- the null space is used for all instruction module and entry context spaces
- the statistics collection and domain exception process models are assigned to the computation cycle of lowest precedence
- the clock is assigned dispatching priority just below that of the error signal process source
- the other R-process sources of the section are assigned the lowest dispatching priorities, in the order they appear in the section.

Space identifiers in the section must always be references to entries in the space definition section when field DBI of the header is zero [Figure 12.1]. If the IDT was generated internally, references to existing B-spaces, rather than space definition section references may be supplied, and field SINT provides the means for the phase to determine the type of reference supplied.

In either case, if a B-space is the referent for an entry context space or instruction module of an R-process model, the initialization phase obtains a descendent M-space for use in connecting the model by the equivalent of a LOAD instruction. The original B-space is then deleted from the system if its control bit in field SDIS is set to 1. If a B-space which appears in the name-pointer pair table is deleted, the B-space pointer in the table is replaced by

SECT	SCCC	DOCC	SDIS
PRCLK		PRIPM	
PRFSX		SINT	
TECTX			
OVSPR			
SCSPR			
DOSPR			
IVSPR			
FXSPR			
ERSPR			

Figure 12.7
Initialization Data Table
Service Process Models Section

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value 3, indicating this is the service process models section of the IDT.
SCCC	1	1	Identifier of the computation cycle for the statistics collection process model.
DOCC	2	1	Identifier of the computation cycle for the domain exception process model.
SDIS	3	1	Bits which control the disposition of B-spaces used as antecedents of instruction module or entry context spaces for R-process models. A B-space is deleted from the system if its disposition control bit is set to 1:

Bit	Controls
0	Not used
1	Entry context space of the time event process model
2	Instruction module space of invalid process model exception process model
3	Entry context space for that model

4	Instruction module space of forced system exception process model
5	Entry context space for that model
6	Instruction module space for error signal process model
7	Entry context space for that model

Field	Offset	Bytes	Description and Use
PRCLK	4	2	Dispatching priority of the system clock as an R-process source.
PRIPM	6	2	Dispatching priority of the invalid process model exception process source.
PRFSX	8	2	Dispatching priority of the forced system exception process source.
SINT	10	2	Significant only if the IDT was generated by a checkpoint, in which case the first thirteen bits of the field define the interpretation of the space identifiers in the remaining thirteen fields of the section. Correspondence between bit and field is obtained by taking the bits from left to right and the space fields from top to bottom; the rightmost three bits are not used. If a bit is zero, the corresponding space identifier is the name of an entry in the space definition section of the IDT. If a bit is 1, the space identifier is a B-space pointer.
TECTX	12	4	Identifer of the entry context space for the time event process model.
OVSPR	16	8	Identifiers of the pair of spaces required to define the overrun process model. The first four bytes identify the instruction module space, the last four the entry context space.
SCSPR	20	8	Identifiers of the pair of spaces required to define the statistics collection process model.
DOSPR	32	8	Identifiers of the pair of spaces required to define the domain exception process model.
IVSPR	40	8	Identifiers of the pair of spaces required to connect the invalid process model exception process model.
FXSPR	48	8	Identifiers of the pair of spaces required to connect the forced system exception process model.
ERSPR	56	8	Identifiers of the pair of spaces required to connect the error signal process model.

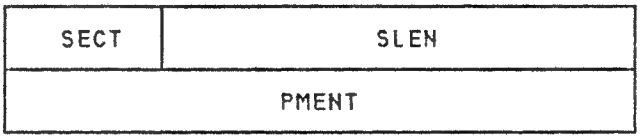

Figure 12.8
Initialization Data Table
Regular Process Models Sections

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value of the identifier which indicates the section type.
SLEN	1	3	Total length of the section in bytes.
PMENT	4	Var	Entries which specify process models to be defined or connected.

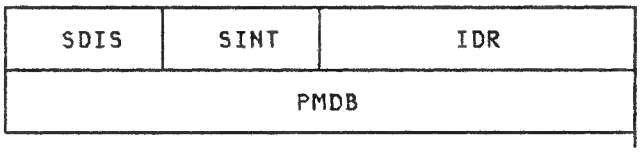

Figure 12.9
Initialization Data Table
Regular Process Model Entry

Field	Offset	Bytes	Description and Use
SDIS	0	1	Bits which control the disposition of B-spaces used as antecedents of M-spaces required for field PMDB of the entry. A B-space is deleted from the system if its disposition control bit is set to 1:

Bit	Controls
0	Instruction module space
1	Exception module space
2	Entry context space
3-7	Not used

Field	Offset	Bytes	Description and Use
SINT	1	1	Significant only if the IDT was generated by a checkpoint, in which case the bits define the interpretation of space identifiers in field PMDB of the entry. If a bit is zero, the corresponding space identifier is the name of an entry in the space definition section of the IDT; if a bit is 1, the space identifier is a B-space pointer. Bit and identifier correspondence is the same as that of field SDIS.
IDR	2	2	Identifier of an R-process source or I/O device to which the process model is to be connected. The field has no significance for a C-process model.
PMDB	4	Var	The RMDB, DMDB, or CMDB which is to be connected or defined.

the M-space pointer to its descendent.

12.7 Regular Process Models

The R-process models, D-process models, and C-process models sections of an IDT are variable in length, with the section format described in figure 12.8. PMENT field entries in each of the three sections have the same basic format, as described in figure 12.9.

If a section is not present in the IDT, the corresponding process models initialization phase proceeds directly to the next phase of initialization. When a section is present the phase connects or defines all process models whose definition blocks are contained in the xection. Space identfiers in ■ block must always be references to entries in the space definition section when field DBI of the header is zero. If field DBI is non-zero, field SINT provides the means for the phase to determine whether the reference is to ■ space definition entry or is ■ pointer to an existing B-space. Field SDIS of each entry determines the disposition of any such B-space, in the same way as does its namesake field in the service process models section.

12.8 System Checkpoint

A system checkpoint generates an IDT whose content allows system operation to restart with conditions restored to those in effect at the time of the checkpoint. As this requires system operation to be suspended during generation of the IDT, and may require substantial output, the effectiveness of the CHECKPOINT instruction (CHECK) is controlled by bit 6 of the error signal mask [Section 11.4]. If the bit is zero, the instruction is effective only when executed within an error signal process; nothing happens if it is executed within an R-process of any other family except to return a condition code indicating that checkpoint is inhibited. If the bit is 1, CHECK is effective within any R-process.

When CHECK is effective, ■ signal is transmitted to all CPU and PPU requesting suspension of operation. The signal will inhibit pending conditions for process sources from becoming effective, and will cause

those CPU not interpreting the CHECK instruction to cease activity at the completion of the next microinstruction sequence which preserves integrity of the local data. PPU instruction interpretation activity will also cease at the first opportunity, but microcode activity will continue, if necessary, until completion of all I/O operation sequences started prior to the reception of the signal. Consequently, all internal activity will eventually stop, leaving the system in some consistent operational state. The CPU interpreting the CHECK instruction will then generate an IDT corresponding either to a full checkpoint or a partial checkpoint, depending on the option selected for the instruction.

- A full checkpoint dumps all M-space and B-space data in the system into the space definition section, with the entries set to restore pointers [Section 12.5].

- A partial checkpoint dumps only the B-space data; the entries can be set either to restore pointers or not, as specified by a second option of the instruction.

The identifier of the output device for the checkpoint IDT is specified by an operand of the CHECK instruction. If the device is a real device, the space definition section of the IDT will contain the appropriate data, and field DBI of the header is set to zero. If the device is the null device, the instruction will be rejected unless a partial checkpoint is specified. In that case, an IDT without a space definition section, but with a non-zero DBI field, will be placed into the initialization area of B-storage.

Apart from the space definition section and the header, the composition of a checkpoint IDT is model-dependent. For some EPSILON systems, the complete system data area is copied into the system checkpoint section, eliminating any requirement for for the system parameters, dispatching structure, service process models, and regular process models data sections. Other systems reduce the amount of data in the system checkpoint section by output of the other sections. In all cases, however, the system checkpoint section contains the state vectors of all processes in the system, and the space allocation list in effect at the time of the checkpoint.

The system checkpoint initialization phase replaces data in the system data area by any corresponding data in the system checkpoint section. Consistency checks to assure that no conflict arises from the substitution are applied at each stage, and failure at any stage will force system termination.

- The space allocation list in the section is compared with the existing space allocation list. If any pointers from previous periods were used in the space definition phase [Section 12.5] the section list must be an exact subset of the existing list; if not, a consistency failure has occurred. If previous pointers were not used, or if there is no inconsistency, the custody, access, and domain data for each spce in the name-pointer pair list is transferred from the section allocation list to the existing list. Spaces and domains in existence at the time of the checkpoint are therefore restored to their original state.

- If the section contains system parameter, dispatching structure, R-process model, or D-process model data, the section data replaces its counterpart in the

system data area.

- The state vector data in the section is then matched against the dispatching data in the system data area. A consistency failure occurs if there is a process state vector corresponding to a process model not assigned to a computation cycle or connected to a source, or if the state vector data conflicts with the family data. If there is no failure, the state vector data is transferred to the system data area.

- If the section contains C-process model data, it must match the existing C-process model data wherever the process model names are the same; a consistency failure is recorded if there is a mismatch for some model. If there is no failure, the data for those process models of the section not already in the existing C-process model list is transferred to that list, and to the system data area where appropriate.

After completion of the system checkpoint phase, the internal state of the system is either the same as it was at the time of the checkpoint, or contains the checkpoint state as a proper subset. However, neither application data, data on I/O devices with removable media, nor devices for which positioning is significant, need be in the state they were at checkpoint. If the system must be restored to full checkpoint condition, the final steps can only be accomplished by application initialization.

12.9 Application Initialization

The application initialization data section of an IDT is variable length, with the format described in figure 12.10.

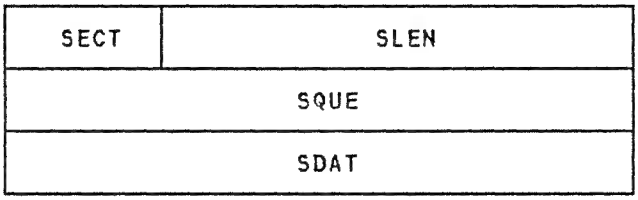

Figure 12.10
Initialization Data Table
Application Initialization Section

Field	Offset	Bytes	Description and Use
SECT	0	1	Contains the value 8, indicating this is the application initialization section of the IDT.
SLEN	1	3	Total length of the section in bytes.
SQUE	4	4	Name of an input queue which is to receive the section data.

SDAT	8	Var	Data to be supplied to the queue for initialization of an application or operating system.

Whether the section is present in the IDT or not, the application initialization phase always completes the initialization procedure.

- If there was no system checkpoint section in the IDT, an IDT with non-zero DBI field in the header is generated and placed into the B-storage initialization area. This IDT, which is the same as would be generated by a partial checkpoint, provides a means for restarting the next operational period with operating condition identical to the period just being started.

- If an application initialization section is present, the equivalent of a QIX instruction applied to the SQUE field is executed. If a queue index is returned, an M-space is allocated to hold the data in field SDAT, and the space is placed into the queue. A warning report will be generated if there is no queue with the specified name.

- Statistical counters are assigned for system data, R-process sources, and I/O devices if counter assignment is not suppressed [Section 10.4].

- The system clock is synchronized with external real time by setting its value to the difference between current time and the standard epoch. The time of day is supplied either by the system operator or by an external timing signal. A basic cycle TERB is generated and inserted into the timing queue.

- The time of initialization is recorded, sequence numbers are reset, and process source inhibition removed. The system is then free to operate normally.

- A warning report is generated if any inconsistency or error was noted during initialization which was not cause for system termination.

An application initialization data section can be output as part of a checkpoint IDT by selecting that option of the CHECK instruction. The data for the SQUE and SDAT fields must have been previously generated and placed into an M-space, in the order required for the section. The fields are located by the second operand of the instruction, and the SDAT field is assumed to extend from its first byte to the last byte of the M-sapace.

12.10 System Termination

System termination is invoked by a DIAGNOSE instruction, or from the operator console by depressing the termination key. Termination triggers the signal which causes the system 12.8]. When activity has ceased, the CPU processing the termination signal will update the space allocation list in B-storage. This will assure that the IDT in the B-storage initialization area will be consistent with a restart from the point of termination. The system will then be reset to the initial operational state.

12.11 Instruction Descriptions

CHECKPOINT

CHECK R1,M3,D2(B2) <RS>

The instruction is terminated with condition code 3 if arithmetic register R1 does not contain the identifier of an I/O device, or the identifier of the null device. It is terminated with condition code 2 if the process executing the instruction is not an error signal process and bit 6 of the error signal mask is zero. If arithmetic register R1 contains the identifier of the null device, bit zero of mask field M3 must be 1 (partial checkpoint), bit 1 of the field must be zero (pointers to be restored), and bit 2 must be zero (no application initialization section). The instruction is terminated with condition code 3 if any of these bit conditions is not met. If bit 2 of field M3 is 1, the instruction is suppressed with a specification exception if the second operand does not define a location on a word boundary.

If the instruction is not terminated or suppressed, a signal is broadcast to all CPU and PPU requesting cessation of activity. The CPU interpreting the instruction then idles until all cessation bits are recorded in the system data area, or until a basic cycle has expired. If a basic cycle expires before cessation of activity has been completed the signal is broadcast again. If the signal is trnasmitted 256 times in succession without activity having ceased or a delay request having been received from a PPU, the cessation request is cancelled and the instruction is terminated with condition code 1. A delay request by a PPU will cause the signal trnasmission sequence count to be reset to 1.

When cessation of activity has been completed, the total length required for the IDT is computed. The length computation takes into account the type of IDT to be generated and the conventions employed for output of system checkpoint data by the model of EPSILON system within which the instruction is being executed. A full checkpoint (bit zero of field M3 set to zero) generates a space definition section containing data for all M-spaces and B-spaces in the system. A partial checkpoint on a non-null device generates a space definition section containing data for all B-spaces in the system. If bit 2 of field M3 is 1, an application initialization section is to be output. The section content, which is located by the second operand, extends to the end of the space in which it is located. A request is made for an M-space to contain the header and the system checkpoint data section. The instruction is terminated with condition code 1 if space is not available.

The header and system checkpoint data section are assembled and placed into the space allocated, for output as a single record. If the output device is null, the record is placed into the initialization area of B-storage. If the device is non-null, an I/O request is placed into its request queue, and a PPU through which the device is attached to the system is signalled to become active. Execution of the instruction is suspended until completion of the I/O request. !tf The remaining data sections of the IDT are assembled, if assembly is required, and output in sequence, each section consisting of one or more physical records. When all sections have been output, a signal is broadcast to all CPU and PPU requesting resumption of activity. The instruction is then terminated with condition code 1 if any error occurred during output of the

IDT, and with condtion code zero if no error was detected.

Process Class: R

Condition Code:

0	IDT output successful
1	IDT could not be output
2	Checkpoint inhibited
3	Device not available

Exceptions:
Specification